

TEKPhysics

TekPhysics

Teaching and Education Kit for Physics

Abstract

A program designed to aid the teaching of real-world physics in an easy-to-understand way. Uses real-time simulation and 3D graphics processing to create an engaging user experience.

Joshua Cowdell-Smith

Candidate number: 3078, Centre number: 20570

Contents

Analysis.....	1
Problem Definition.....	1
Background Research.....	1
Scrap Mechanic.....	1
Kerbal Space Program.....	4
OpenGL.....	5
Complex Algorithms.....	7
Data Input and Output.....	12
End User.....	13
Potential End Users.....	13
The End User Interview.....	14
Interview Analysis.....	15
Objectives.....	15
Modelling.....	19
Two Dimensional Physics.....	19
Three Dimensional Graphics.....	35
Design.....	42
Mission Statement.....	42
Project Overview.....	43
User Interface.....	43
User Interface Layout.....	43
Main Menu.....	45
Builder Menu.....	46
Runner Menu.....	47
Save Menu and Load Menu.....	48
Required Libraries.....	49
Physics vs Graphics.....	49
TekPhys.....	50
Threading and Thread Communication.....	50
Physics Engine Loop.....	52

Parsing 3D Model Files.....	53
Velocity and Angular Velocity Integration.....	57
Data Structures.....	57
TekPhys Key Algorithms.....	59
Collider Structure Generation.....	59
Rigid Body Properties Generation.....	62
Triangle-Triangle Collision Detection (GJK).....	64
Triangle-Triangle Collision Contact Points (EPA).....	68
Triangle-Oriented Bounding Box (OBB) Collision Detection.....	74
OBB-OBB Collision Detection.....	76
Collider-Collider Collision Detection.....	78
Rigid Body Collision Response.....	81
TekGL.....	84
GLFW Interface.....	84
OpenGL / GLAD Interface.....	85
Meshes.....	85
Shaders.....	85
Camera.....	86
Textures.....	86
Text / Font.....	87
Materials.....	90
Entities.....	91
TekGui.....	91
Button.....	92
Text Button.....	92
Text Input.....	92
Window.....	93
List Window.....	93
Option Window.....	93
File Structures.....	95
Project Structure.....	95

Core Folder Structure.....	97
TekGL Folder Structure.....	98
TekPhys Folder Structure.....	99
Res Folder Structure.....	100
Windows Folder Structure.....	101
Shader Folder Structure.....	102
Tests Folder Structure.....	103
Function Listing.....	104
TekGL.....	104
Main.....	104
TekGUI.....	110
Primitives.....	111
List Window.....	113
Box Manager.....	114
Window.....	114
Text Button.....	116
Button.....	117
Option Window.....	119
Text Input.....	124
Thread Queue.....	126
Vector.....	127
Queue.....	128
Bit Set.....	129
YAML.....	130
Stack.....	135
Hash Table.....	135
File.....	137
Exception.....	138
Priority Queue.....	139
Test Suite.....	139
Text.....	141

Mesh.....	142
Manager.....	144
Entity.....	147
Font.....	148
Texture.....	149
Material.....	149
Camera.....	150
Shader.....	151
Engine.....	153
Geometry.....	155
Body.....	156
Collisions.....	157
Collider.....	162
Scenario.....	164
Hierarchy Chart.....	167
Technical solution.....	216
Guide to Complex Functions.....	216
Source Code.....	217
main.c.....	217
tekg1.h.....	272
core/bitset.c.....	273
core/bitset.h.....	280
core/exception.c.....	281
core/exception.h.....	286
core/file.c.....	287
core/file.h.....	291
core/hashtable.c.....	291
core/hashtable.h.....	304
core/list.c.....	305
core/list.h.....	315
core/priorityqueue.c.....	316

core/priorityqueue.h.....	319
core/queue.c.....	320
core/queue.h.....	323
core/stack.c.....	323
core/stack.h.....	326
core/testsuite.c.....	326
core/testsuite.h.....	326
core/threadqueue.c.....	328
core/threadqueue.h.....	332
core/vector.c.....	332
core/vector.h.....	339
core/yml.c.....	340
core/yml.h.....	382
tekgl/camera.c.....	383
tekgl/camera.h.....	387
tekgl/entity.c.....	387
tekgl/entity.h.....	392
tekgl/font.c.....	393
tekgl/font.h.....	402
tekgl/manager.c.....	403
tekgl/manager.h.....	415
tekgl/material.c.....	416
tekgl/material.h.....	430
tekgl/mesh.c.....	431
tekgl/mesh.h.....	443
tekgl/shader.c.....	444
tekgl/shader.h.....	453
tekgl/text.c.....	453
tekgl/text.h.....	463
tekgl/texture.c.....	464
tekgl/texture.h.....	466

tekgui/box_manager.c.....	466
tekgui/box_manager.h.....	471
tekgui/button.c.....	472
tekgui/button.h.....	479
tekgui/list_window.c.....	480
tekgui/list_window.h.....	491
tekgui/options.yml.....	491
tekgui/option_window.c.....	493
tekgui/option_window.h.....	527
tekgui/primitives.c.....	529
tekgui/primitives.h.....	538
tekgui/tekgui.c.....	539
tekgui/tekgui.h.....	551
tekgui/text_button.c.....	552
tekgui/text_button.h.....	559
tekgui/text_input.c.....	560
tekgui/text_input.h.....	574
tekgui/window.c.....	575
tekgui/window.h.....	587
tekphys/body.c.....	588
tekphys/body.h.....	597
tekphys/collider.c.....	598
tekphys/collider.h.....	618
tekphys/collisions.c.....	619
tekphys/collisions.h.....	673
tekphys/engine.c.....	674
tekphys/engine.h.....	690
tekphys/geometry.c.....	692
tekphys/geometry.h.....	700
tekphys/scenario.c.....	701
tekphys/scenario.h.....	717

shader/button.glvs.....	718
shader/fragment.glfs.....	719
shader/image_vertex.glvs.....	720
shader/image_fragment.glfs.....	720
shader/line_vertex.glvs.....	720
shader/line_fragment.glfs.....	721
shader/oval_vertex.glvs.....	721
shader/oval_fragment.glfs.....	721
shader/texture.glvs.....	722
shader/texture.glfs.....	722
shader/text_vertex.glvs.....	723
shader/text_fragment.glfs.....	724
shader/vertex.glvs.....	724
shader/window.glvs.....	724
shader/window.glg.....	725
shader/window.glfs.....	727
Testing.....	728
Test Strategy.....	728
Test Plan – Unit Test.....	728
Test Plan – Video.....	737
Evaluation.....	748
Objective Evaluation.....	748
End User Feedback.....	758
Interview.....	758
Interview Evaluation.....	760
Future Improvements.....	760
Air Resistance Simulation.....	760
Increased Model Customisation.....	762
Simple Scripting Abilities.....	763
Appendices.....	765

Analysis

Problem Definition

Within physics education, there are very few visual methods to aid the explanations of complex topics. Often, we are limited to simple drawings or diagrams, and in some rare cases simple two-dimensional demonstrations are available online. To see real, three-dimensional physics in action, we are left to conduct practical experiments, which are time-consuming, expensive, and are limited by school budgets and lesson timetables. This may leave students confused, and without a proper mental picture of how physics works in reality. Additionally, students may feel dissatisfied with their physics education, feeling as if their studies do not apply to reality.

The aim of this project is therefore to create an application that can help students to visualise complex problems within mechanics by providing access to simulated visual and numerical data for a variety of physics related problems. The application should be capable of simulating and displaying complex scenarios involving multiple objects and forces in real time. It should also come with some pre-existing scenarios that can be simulated, while still allowing the user to construct their own if needed.

By solving this problem computationally, we avoid the need to purchase and use expensive equipment, and also avoid the risks involved with practical experiments that may involve projectiles and heavy weights. It also allows us to experience situations which may not be feasible to reproduce in a classroom, for example simulating the forces involved in colliding heavy objects such as cars, and understanding how the forces scale with the increase in size of the objects.

Background Research

Scrap Mechanic



Figure 1: A screenshot of the game 'Scrap Mechanic'

Scrap Mechanic is a sandbox game that allows players to build vehicles, machines and other structures, with a strong emphasis on engineering, construction and survival. Scrap Mechanic is a game I have enjoyed for many years, and is the main inspiration for my NEA. There are many aspects that I would love to implement in my game, while some things I would like to overhaul.



Figure 2: A player constructing a creation by dragging out a block of material.

One of the features of scrap mechanic is the ability to construct creations using blocks by clicking and dragging to fill areas with material. There are also custom parts such as wedges, wheels and pipes that make creations look more interesting. Additionally, Scrap Mechanic gives you access to parts such as bearings, pistons and thrusters that allow for a multitude of ways to power creations. Furthermore, Scrap Mechanic gives you access to parts such as controllers, sensors and logic gates, which allow you to make complex creations which can react to their environment.



Figure 3: Collision detection and response in Scrap Mechanic

Scrap Mechanic also has a realistic physics engine, which makes building creations more rewarding and challenging, as the creations must abide by the laws of physics. This also leads to more immersive gameplay, as the creations feel more real. Players must account for weight, balance and movement when designing different vehicles.

In the game, there are also three game modes: creative mode allows for unlimited experimentation and resources, to build creations to your heart's content, survival forces you to collect food and resources to build machines and survive, and challenge mode gives you a task to complete with limited parts to build with.

In my implementation, the focus will be on understanding the physics, so I have chosen not to include a survival or challenge mode, opting for just a creative mode. I would also like to implement a similar building style, where creations can be constructed from simple parts that are joined together. I also think that it would be beneficial to add different sensors such as timing and distance sensors, this would allow for physics experiments to be implemented into the game.

Kerbal Space Program



Figure 4: A screenshot of the game 'Kerbal Space Program'

Kerbal Space Program (KSP) is a simulation game, where the player constructs different spacecraft in order to explore space, earn money and collect scientific data. This is another game I have enjoyed for years, and has also inspired this project.

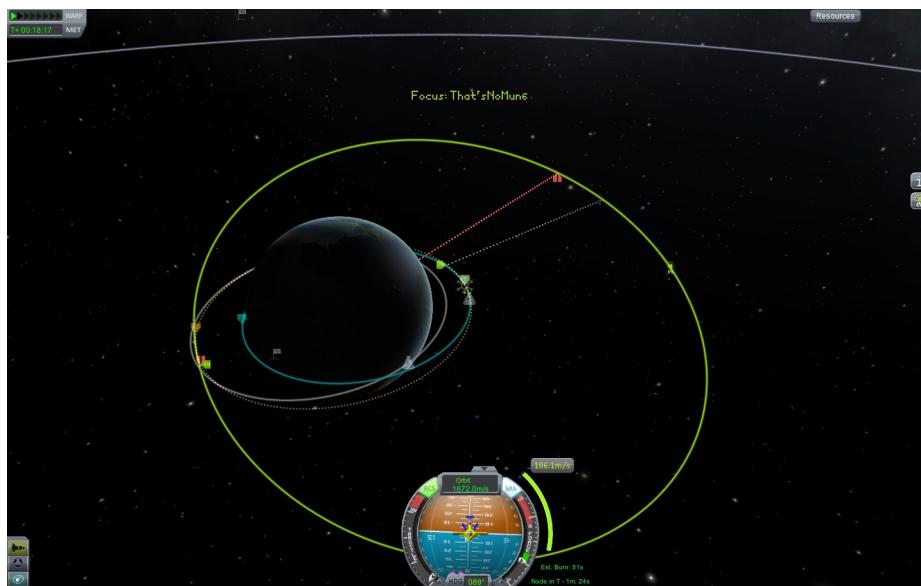


Figure 5: A screenshot showing how different orbits can be viewed in the game

KSP gives players access to parts such as cockpits, fuel tanks, thrusters, wings, parachutes and landing gear in order to construct a range of vessels, including rockets, planes, or mix them into space shuttles. There is unlimited creativity, which makes the game both challenging and rewarding to play.

The physics engine of KSP is able to simulate the forces of gravity and thrusters, as well as orbital mechanics. This allows players to enter orbits around different bodies, and perform transfers between planets. KSP also gives the player

information about the velocity of their craft, the delta-v (amount of velocity change possible given the fuel and weight of the craft), the mass, and much more. KSP also simulates air resistance accurately, allowing for lifting force and drag forces, as well as overheating when going too fast through the atmosphere.



Figure 6: The 'navball' in KSP that shows players information about their creation

In my project, I would expand on the metrics that are given to the user, and also display things like the acceleration, angular velocity and also display the forces acting on objects. This would make the game more useful as a physics simulation. While KSP's implementation of aerodynamics and orbital mechanics is fascinating, I think that it would be out of the scope of this project, and I will instead opt for a simple air resistance model based on cross sectional area and speed.

OpenGL

In order to render my physics simulation in three dimensions, I will need to make use of a graphics API. I have chosen to use OpenGL, as it is cross platform, customisable, well documented, and relatively simple to use compared to other APIs such as Vulkan.

OpenGL is an application programming interface (API) that provides a large set of graphics related functions. This includes functions to allocate video memory on the GPU, to compile and use shader programs, and to render a buffer of pixels to a screen.

OpenGL is simply the standard for functions that should be available to use, however each operating system may implement these differently. Therefore, we also need to use a window creation library (GLFW) to create and manage the

window into which OpenGL will render the simulation, and we need a library to find the operating system specific OpenGL functions (GLAD).

The premise of OpenGL is a system of creating buffers or objects, and binding a single object at a time that you wish to perform operations on. For example, to create a vertex buffer (an array that stores data for the vertices of an object), we need to do something like:

```
1 // written in C (some details omitted)
2 // store the vertex buffer id as a variable
3 unsigned int vertex_buffer;
4 // ask OpenGL to find an unused buffer id, keep this in the
variable
5 glGenBuffers(1, &vertex_buffer);
6 // bind this id as an array buffer
7 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
8 // fill the buffer of this id
9 // notice how we don't use the id after in this call, OpenGL
10 // will always modify the bound buffer until we unbind it
11 glBufferData(GL_ARRAY_BUFFER, sizeof(data), data,
GL_STATIC_DRAW);
```

OpenGL uses a process called a graphics pipeline, which means there are several stages of graphics processing that lead to the final output onto the screen.

The first stage of this is vertex processing. On the programming side, this is where we need to specify a vertex shader. This is a program that is loaded onto the GPU, and is responsible for processing the raw vertex data that we will store in a buffer. This vertex data includes coordinates in 3D space, but could also include the colour, texture coordinates, and vertex normals (a vector pointing straight out of the shape) which are needed to compute the lighting of the shape later in the pipeline. We can also modify vertices, for example translating and rotating them into their correct position in the world (the model may be centred around (0, 0, 0), when in reality it should be able to move around the world), which is done using a 4x4 translation and rotation matrix respectively. We also need a 4x4 projection matrix which is responsible for defining the camera's orientation and field of view. The resultant vertices from this will then be assembled into primitive shapes, which will typically be individual triangles as they are simple to draw and can be used to create any 3D shape, however we can specify other primitives such as lines, or connected triangles.

During the vertex processing stage, we can also optionally specify a geometry shader, which can take single vertices, and use them to generate more primitives that are not stored in the vertex buffer. This can be useful to draw more dynamic structures, or to reduce the size of the vertex buffer.

After this, the vertices will undergo processes such as clipping (culling vertices that are not visible on the screen to reduce drawing time), back face culling (removing faces that are behind the camera), as well as the perspective divide and the viewport transform which will convert the points in 3D space and place them into the correct position on the screen.

These vertices will then be rasterised using a fragment shader. This is responsible for determining how each pixel of the primitive is coloured. The previous values we had at each vertex (colour, texture coordinates etc.) will be linearly interpolated across the primitive. For triangles this involves the use of barycentric coordinates, essentially how much of each vertex of the triangle each point inside of it is worth. We also perform per-pixel lighting calculations such as ambient and diffuse lighting at this stage.

This pipeline allows us to draw a variety of different objects to the screen. Aside from 3D objects, we can alter the vertex shader to allow us to draw 2D shapes, and using textures also allows us to render text and other images as part of a user interface system.

Complex Algorithms

As my NEA project revolves around simulating physics, naturally a lot of equations are required in order to properly simulate this. A lot of the complexity arises due to the jump from 2D to 3D simulation, as this creates another axis of movement that we need to account for. However, to fully explain the algorithms needed for my project, I will begin with a 2D simulation, and then see how this transfers into the third dimension.

Imagine a 2D object being thrown from a cliff face.

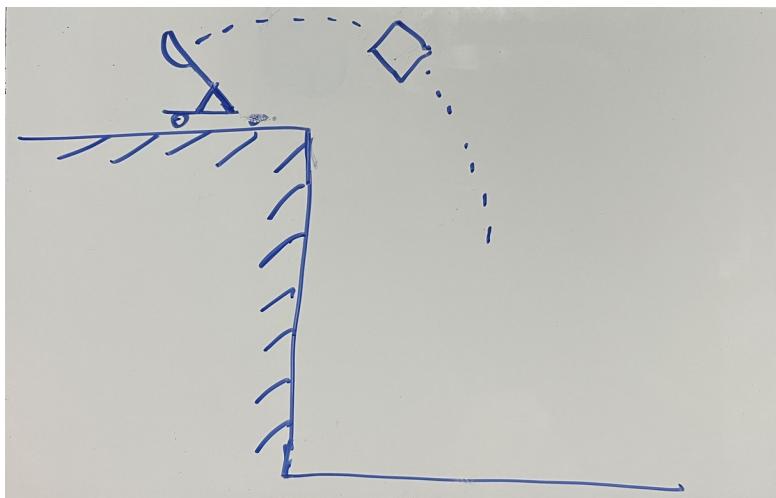


Figure 7: An object being thrown from a cliff face.

There are several forces at play during this event. We must consider the acceleration due to gravity that happens on the object. The weight (W) of the object is equal to the product of its mass (m) and the acceleration due to gravity (g). We also know that due to air resistance, there will be a drag force opposing

the movement of the object. We also know that drag is proportional to the square of speed. From this, we can find the total resultant force, and calculate the change in velocity.

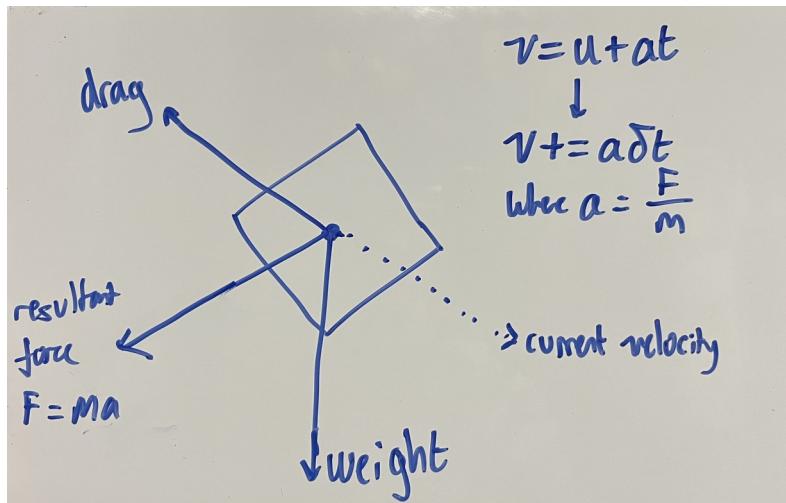


Figure 8: A diagram showing the forces at play on an object falling in air.

In reality, objects will experience a continuous change in velocity, however this process is not possible to mirror on a computer. Instead, physics engines use a time step (Δt) to instead only calculate the state of objects at certain moments in time. Engines can choose to use a fixed time step, or a variable time step. Fixed time step has the advantage of being perfectly predictable given the same starting scenario, though can start to slow down under heavy load. Variable time step has the advantage of running at the same speed regardless of load, however becomes extremely unstable if Δt becomes too large. For physics simulations, fixed time step is almost certainly the way to go, but we can get the best of both worlds by allowing the graphics to run on a variable time step, and linearly interpolate between physics states between frames to give a smooth appearance of motion.

With this in mind, we can calculate the acceleration of our object using $F = ma$, and then find the change in the velocity as $\Delta v = a\Delta t$. However, this only accounts for the forces that act through the centre of mass of the object. Forces that do not act through the centre of mass will create a moment, thus causing a rotational force on the object. This can be seen most notably when the object collides with the ground.

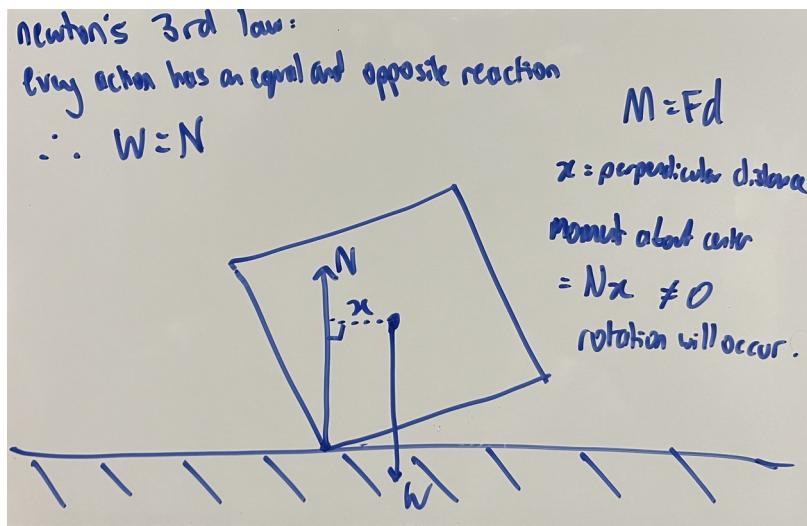


Figure 9: An explanation of how a moment can be created by a collision.

When this happens, we need to update the angular velocity (ω). This can be done by finding the angular acceleration (α) which is equal to the torque (τ) divided by the moment of inertia (I). This makes sense; the torque is the angular force applied, and the moment of inertia represents how hard it is to rotate that particular object. As a result, you essentially arrive at Force divided by an "angular mass" which naturally would give us angular acceleration. We can then use this to find the angular velocity by multiplying by our physics time step (Δt) and adding this to the previous angular velocity.

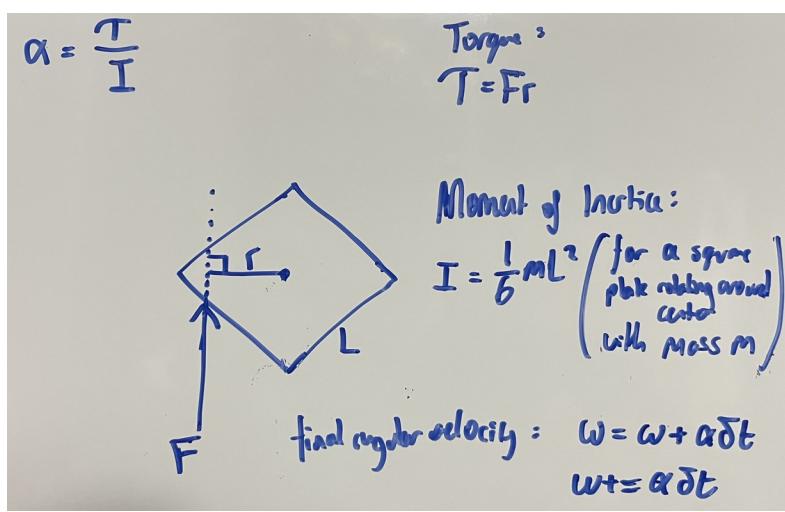


Figure 10: A diagram to show how angular velocity is calculated.

In order to perform these calculations, we also need to know if and where the object has collided with the ground. This is trivial for our square object and a horizontal ground – simply check if the y coordinate is below a certain level.

However, once we implement rotation in the object, or even change its shape entirely, then this approach may no longer work. One of the solutions to this is the Separating Axis Theorem (SAT), which can determine if any two convex polygons are intersecting. This works by taking the normal of each of the faces of the polygon, and then projecting (you can think of this as casting a shadow) both polygons onto this normal. Do this for every normal of every face of both shapes. If the “shadows” of the polygons do not intersect along one of the normals, there is no intersection. However, if every projection overlaps, then that means the polygons intersect. This approach can be expanded to 3D without any difference, we just use three dimensional vectors instead.

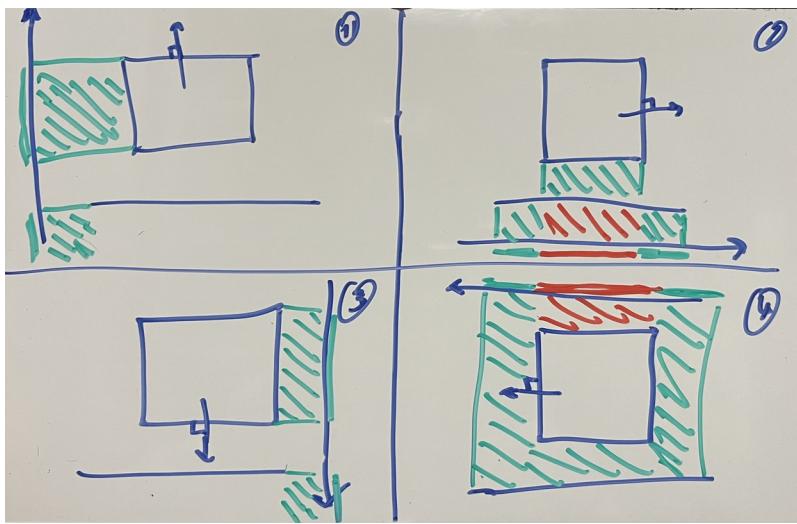


Figure 11: A diagram showing the SAT in action. Green shows non-overlapping areas, red shows an overlap.

This algorithm is typically not used in physics simulation, and the Gilbert-Johnson-Keerthi algorithm (GJK) is used instead as it is a faster algorithm. This could be an option to implement if the SAT algorithm is too slow – collision detection is usually the most computationally expensive part of physics simulation.

Another way to optimise collision detection is by using Oriented Bounding Boxes (OBBs), which are essentially rotated cuboids that surround complex shapes. OBBs are organised in a binary tree, where each OBB contains two sub OBBs. The leaves of the tree are the final faces (in 3D these are the triangles) that make up the object. This improves the time to check for collision detection – if the external OBB does not collide we can ignore the object entirely, rather than computing collisions for each and every face that makes up the object. If a collision is found, then we still only need to check the two child OBBs for a collision. This method can save huge amounts of processing time, which is valuable in a real-time simulation.

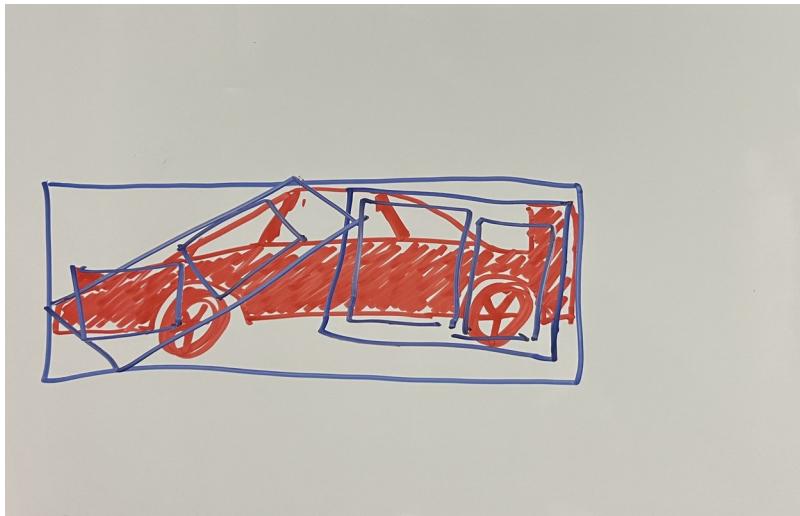


Figure 12: An inaccurate example of how an object may be divided into a tree of bounding boxes.

Just knowing where a collision occurs is not enough for our physics simulation. We also need to calculate the forces involved when a collision happens. The first force we need to think about is the impulse (J) applied at the point of the collision. The impulse can be calculated using the following equation:

$$J = \frac{-(1+e)(v_2 - v_1) \cdot n}{\frac{1}{m_1} + \frac{1}{m_2} + \frac{(r_1 \times n)^2}{I_1} + \frac{(r_2 \times n)^2}{I_2}}$$

v_1, v_2 = velocities r_1, r_2 = distance to
 m_1, m_2 = Masses point of impact
 e = coefficient of restitution from centre of mass
 I_1, I_2 = moments of inertia

Figure 13: An equation to find the impulse of a collision.

To clarify, the coefficient of restitution (e) is a value from 0.0 to 1.0 that represents how 'bouncy' a collision is – a value of 0.0 means that none of the initial velocity is retained following the collision, while a value of 1.0 means that all of the initial velocity is retained. The value is dependant on the material that the body is made of – a sponge would have a low value, while a bouncy ball would have a high value. The values r_1 and r_2 represent vectors from the centre of mass of each object to the point of collision, for example:

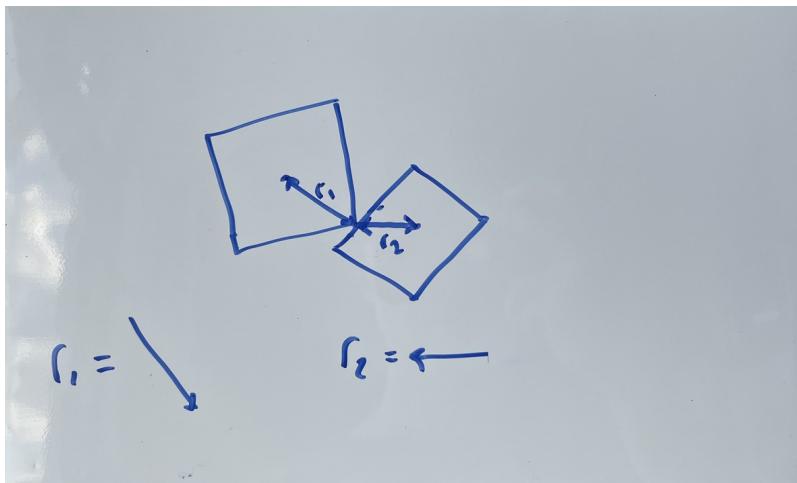


Figure 14: A visual explanation of the two 'r' vectors

Once we have found the impulse (J), we can use it to calculate the acceleration and angular acceleration of the objects. To do this, we apply a force J to one object, and $-J$ to the other object. We have previously defined that $F = ma$, allowing us to calculate the acceleration of the object as F divided by m . Additionally, we can find the angular acceleration, as we know angular acceleration is torque divided by the moment of inertia. We can calculate torque as $\tau = Fr$, where F is our force (the impulse) and r is the distance from the centre of mass, or just the magnitude of our r vector used in the prior calculations.

This is all that is needed to model rigid bodies, whether working in 2D or 3D. However, there are some other things that could be implemented, such as axles/motors, springs, ball and socket joints and hinges. These all essentially work by constraining the movement of a rigid body within some parameters. An axle is limited to rotate in one axis, a ball and socket joint is constrained to rotate about a single point, and so on. This can be achieved using a Jacobian matrix, which essentially computes how much the velocity of a body should change, depending on the constraints that we give it. We can then compare the velocity that we should have, and the velocity that we actually have, and apply an impulse to the body to correct for the difference.

Data Input and Output

Users will input data using the keyboard and mouse to interact with the world. They will be able to move around using the arrow keys or W, A, S and D keys, and look around using the mouse. They will have a menu available to create and build different objects, possibly using primitive objects such as cubes to build larger structures. There will also be a variable slider input to change the speed of the physics simulation. This will allow the user to visualise the physics simulation in slow motion. Aside from things like an options menu and saved games menu, where the user might need to input simple data such as names or sliders / buttons, there is not much else.

The user will also be able to input some starting conditions of the model, for example the initial velocities of each object, their starting positions, their masses and so on. They will then have access to a 'run simulation' button that will begin the simulation of physics. They will then be presented a 'stop simulation' button that can be used to stop the simulation, and return the objects to their initial locations and velocities, so that the simulation can be easily re-run using different values to experiment with the effects of different variables on the simulation.

The program will primarily output a 3D visualisation of the user-created objects to the screen. However, to make the game more accessible to use in classrooms, it will also have the ability to output the data contained in each object, for example its velocity, acceleration, mass and so on. Depending on the user's configuration of the settings, it may also be possible for the program to output a simulation of the lighting of the scene the user is in, whether the user has created a light-emitting object or whether the scene is lit by ambient light / sunlight. The main purpose of the program is not to output an accurate simulation of light, but just as a visual aid to make the simulation appear more realistic.

End User

Potential End Users

This project is aimed primarily for educational settings, to be used either by teachers or students, and allow for a much more visual way of seeing and understanding how physics works in reality. The program can see use in many areas of education, either in Physics, or in Maths when studying Mechanics. Features such as varying the rate of time passing, clicking objects to view their properties such as mass and velocity, and being able to construct scenarios based on their imagination or to model different problems.

For teachers, the program could be displayed on a projector or screen in front of the class to visualise a problem, and therefore demonstrate how it can solved. It would serve as a replacement for drawing complex diagrams that can be hard for students to interpret.

More broadly, schools may find use of the program in low-budget settings, where resources to set up and conduct practical experiments is limited. Instead of spending an entire lesson, and possibly having to pay to replace equipment if damaged, an experiment can be shown using the program as a quick activity.

For student, the program can become a creative outlet to design different scenarios that they have imagined. Perhaps they have wondered the force produced by dropping an elephant from the top of a 100 storey building when it hits the ground, or the terminal velocity it reaches while falling.

Students can also use this as a way of self-teaching, and helping themselves to understand how the physics and calculations relate to reality, without needing a

teacher to explain it or draw diagrams for them if they are unsure, and without worrying about making incorrect assumptions on a problem.

The End User Interview

I decided to speak to a potential user of my software to gain a deeper understanding of the needs of my clients. I chose my mechanics teacher, Stuart Cliff, to interview, and I asked him the following questions:

Many students find mechanics / motion hard to understand. What do you think is the main reason for this?

I think we can look around and see things but most people don't get that deep understanding of why things happen. You can go on a roller-coaster and go "that was fun", but most people don't sit there and go "I wonder which way the acceleration is pointing at each point". And you need to start to try and view the world in that way, to understand why things work. And some people have just 'got it', you get some I call 'natural engineers', they can just see something and go "that's happening because of this", but the majority of the population aren't like that.

Do you also find it difficult in terms of teaching this topic? Why or why not?

I don't, mainly because I've been doing it for so long, but you do sometimes find yourself having to quickly on your feet think of a nice way of explaining something. You've got a student who's really struggling with a misconception, and you have to then go "well, this is why it happens". And I tend to move my hands around a lot, I use pens, I use whatever I've got to hand to try and demonstrate it in real life. I think one of the big things if we had time to do in Maths would be actual experiments, and years ago we did, we'd have pulleys and things like that, we'd have masses in the cupboard. If you go in the back of Mark's cupboard downstairs it's all in there, we just don't have the time to do it.

How would you feel about a visual program to help demonstrate physics scenarios?

I would love a visual program to demonstrate it, we used to have one... I can't remember the name of it, I'll try and find it if I can.

What kind of user interface would be most intuitive?

Graphical, draw it. If I want a box, draw a box. If I want a force, I want to be able to drag an arrow in the direction that the force is going. Or, for ease of use you might want to be able to actually input it as a vector as well. So ideally something in between the two.

Some potential features could be: displaying force arrows; displaying mass, velocity, acceleration on the screen; pausing or slowing time. What do you think of these features, and are there any more features you would like to see?

Yep, displaying force arrows, displaying mass, yep. Being able to export, and if you've got something that does move, automatically draws a graph, that kind of option that's what you want.

How much customisation would you need in terms of the objects or scenarios that are available? Where is the balance between entirely preset scenarios and entirely user-created scenarios?

I'd like the ability to have some preconfigured ones, because it makes your life easier obviously, but you should be able to set up a scenario yourself.

What features do you think students would appreciate when using the program themselves?

It'd probably be more useful as a classroom thing, so I could go (pointing at something demonstrated earlier in the lesson) that setup there that we've got, I could draw that on the computer, click start, and it will move as it should in real life, or in this case obviously not as it's balanced. And you should be able to change where 'x' is (this was a variable in the demonstrated problem) and cause it to actually move live in front of you.

Do you foresee any drawbacks of the program might exist that would make you avoid using it?

Until we test it who knows, basically. There have been some attempts, like I said, there was an Irish company, I can't remember the name, and it was really, very good, it was just very expensive. If we wanted it on every computer in the maths department, we're talking about £2000, which for educational software it's too much. No one's going to pay for that.

Interview Analysis

From the interview, several key features that are needed in the program have been identified. Firstly, many students struggle to understand mechanics problems as they don't have an intuitive sense of what's happening – as a result the program should be able to simply and intuitively display what is happening with a graphical user interface. Most significantly, the program should be able to display a problem to the user by rendering the objects being simulated and the direction of forces acting on the objects. The users should also be able to modify the conditions of the simulation easily by using the mouse to drag objects and forces to where they are required. Additionally, the program needs to be able to produce a graph or table of data from the simulation to allow for numerical analysis of a problem if needed.

Objectives

1. Construct some programming utilities for the application
 - a. Create an exception handler
 - i. It should be able to record different types of exception.

- ii. It should pause the current function if an exception occurs inside of it.
 - iii. It should record a trace of all function calls that led up to the exception.
 - b. Create a file utility
 - i. It should be able to read data from files.
 - ii. It should be able to write data to files.
 - c. Create a YAML file utility
 - i. It should be able to read a YAML file into a tree data structure.
 - ii. Each node of the tree should store the name of a key/identifier in the YAML file.
 - iii. Each node should have a child, which should either be another node of the tree, or data such as a string, number, or list of values.
 - iv. There should be functions to access the stored data using the string keys.
- 2. Construct a rendering engine for the application
 - a. The engine should perform a wide range of functions in two dimensions
 - i. It should be able to render primitive shapes such as circles or polygons.
 - ii. It should be able to render text at different font sizes.
 - iii. It should be able to render anti-aliased text.
 - b. The engine should perform a wide range of functions in three dimensions
 - i. It should be able to load the information needed to draw a three-dimensional shape from a file.
 - ii. It should be able to draw a three-dimensional representation of the object onto the screen, based on the position of the camera.
 - c. The engine should be able to perform lighting calculations
 - i. It should be able to use different lighting effects that will vary the brightness of objects in the world based on light sources.
 - ii. It should be able to load different shader programs (programs that determine the colouring of objects) from a file.
 - d. The engine should allow for different materials to be attached to objects
 - i. Each material should have an associated shader program.

- ii. Each material should have an associated colour, stored as 3 floating point numbers representing redness, greenness, and blueness in a range from 0.0 to 1.0
- iii. Each material should have (if required) an associated image file that serves to texture it's object with a pattern other than a flat colour.
- iv. Each material should also have (if required) an associated image file that represents normal vectors on the face of an object in terms of the perpendicular x, y, and z coordinates. This will be used to change how lighting reacts to an object on different parts of a flat surface.

3. Construct a physics engine

- a. The engine should accurately depict real-life physics
 - i. The engine should work by simulating physics in small intervals of time.
 - ii. The engine should use a fixed time interval, allowing it to perform the same calculations regardless of hardware speed and other factors.
 - iii. The engine should use equations from Newtonian physics to predict the motion of objects between time intervals.
 - iv. The engine should use respond to colliding objects by applying a corrective impulse to displace the colliding objects.
- b. The engine should work in real-time
 - i. The engine should use a time interval of less than 0.05 seconds (20 intervals per second).
 - ii. The physics engine should run independently of the rendering engine, allowing it to run at the correct pace even if the rendering is slowed.
- c. The engine should be optimised for fast running
 - i. The engine should use a tree of oriented bounding boxes that contain objects in increasingly small containers, so that collisions can be checked using a binary search rather than testing each face individually.
 - ii. The engine should use a well-tested triangle collision algorithm such as GJK + EPA
 - iii. The engine should employ the separating axis theorem to test collisions between oriented bounding boxes.

- iv. The engine should use a broad and narrow collision phase to avoid unnecessary computation.
4. Construct a user interface to allow users to control the program
- a. It should be easy to program new areas of the user interface (UI)
 - i. There should be UI elements such as containers, text, buttons, value sliders and tick boxes.
 - ii. UI elements should be modular, meaning they can be combined together to make a full user interface.
 - iii. Interactive UI elements should allow the programmer to access their stored values.
 - iv. It should be possible to store a UI configuration in a file, so that the UI can be modified without rewriting any code.
 - b. It should be simple for the user to navigate the UI
 - i. The UI should have labelled buttons that can be pressed to perform different actions within the simulator.
 - ii. The UI should have short labels (less than 3 words) for buttons or other inputs, to help keep the UI concise.
 - iii. The UI should contain help buttons that give a description of the functions of each button or input.
 - c. The UI should change relevant to the state of the simulations
 - i. The UI should have a main menu that allows users to select a simulation mode (pre-existing scenarios, or a scenario builder) or to edit the settings
 - ii. The UI should be able to change to a settings panel when the settings button is pressed, which will allow the user to configure the graphics settings and physics simulation settings.
 - iii. The UI should be able to change when the scenario builder mode is activated, to allow scenarios to be created or updated
 - iv. The UI should be able to change when the simulation is being run, to display the simulation running in real time.
 - d. The UI should be effective during the creation of new scenarios
 - i. The UI should allow users to quickly edit objects' masses, initial velocities, constants such as friction and restitution, colour and shape.
 - ii. The UI should be operable using the mouse to drag objects or points on objects to edit them.

- iii. The UI should allow for changes to be undone and redone, making use of buttons or the shortcuts Ctrl-Z and Ctrl-Y respectively.
 - iv. The UI should allow objects to be copied, cut, and paste in the world, making use of buttons or the shortcuts Ctrl-C, Ctrl-X, and Ctrl-V respectively.
 - v. The UI should allow the simulation to be re-run following any changes to the starting conditions.
- e. The UI should be effective during the running of scenarios
- i. The UI should display time control functions such as play and pause, which will allow and block the passage of simulation time.
 - ii. The UI should display time control functions that change the speed at which time advances, so that physics can be viewed in slow motion.
 - iii. The UI should also allow time to be experienced in reverse, by caching the states of objects at a steady time interval and then displaying their progression in reverse chronological order when requested.
 - iv. The UI should have the option of exporting simulated data as a spreadsheet file (.csv) for further processing if the user needs the raw data.
 - v. The UI should be able to display graphs such as velocity-time graphs that represent the motion of different objects.

Modelling

Two Dimensional Physics

In order to test the validity of this project, I wrote some code to prototype my application. I wanted to explore some of the mathematics and algorithms behind a physics simulator, so to begin I created a simple 2D collision simulator which can check for a collision between two polygons.

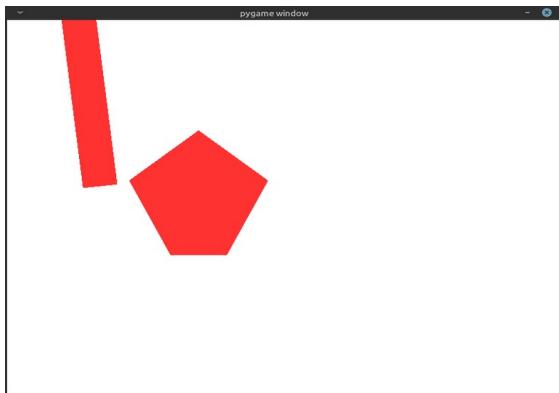


Figure 15: Two polygons just before a collision

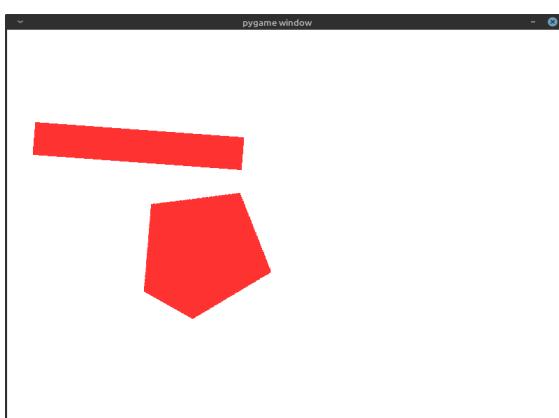


Figure 16: Two polygons moments after a collision

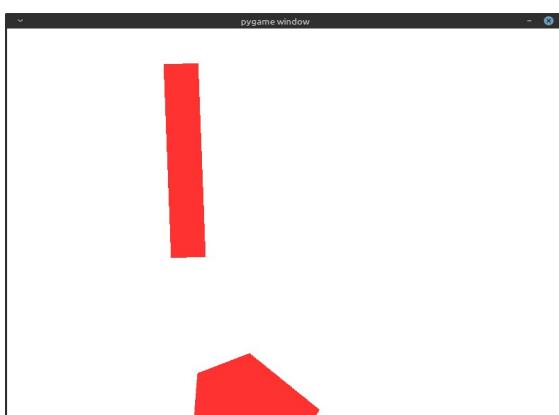


Figure 17: Two polygons a few moments after the collision

Additionally, I have uploaded a short clip onto YouTube of this prototype working, which can be accessed via this link:

<https://www.youtube.com/watch?v=YX84gK4LRv8>

Or via this QR Code:



Below is the full source code for this prototype.

```

1 import math
2 import time
3 from abc import abstractmethod
4
5 import pygame
6 import pygame.math as pm
7
8 # abstract physics object with just a step method
9 class AbstractObject:
10     @abstractmethod
11     def step(self, timestep):
12         pass
13
14 # lists of objects, and their object group
15 objects: list[AbstractObject] = list()
16 objects_group = pygame.sprite.Group()
17
18 # method to run physics step
19 def physics():
20     timestep = 1 / 60
21     for phys_object in objects:
22         phys_object.step(timestep)
23
24 # method to find nearest point on a line segment
25 def nearestPoint(line_a: pm.Vector2, line_b: pm.Vector2, point: pm.Vector2):
26     # B
27     # X==P
28     # | /
29     # | /
30     # A
31
32     # given A, B and P, find X
33
34     # get vectors AB and AP
35     ab = line_b - line_a
36     ap = point - line_a
37     # project AP onto AB
38     projection = ap.dot(ab)
39     length = ab.length_squared()

```

```

40
41     # find proportion 0.0 - 1.0 that AP takes up of AB
42     dist = projection / length
43
44     # before point A
45     if dist <= 0:
46         contact_point = line_a
47     # after point B
48     elif dist >= 1:
49         contact_point = line_b
50     # somewhere between
51     else:
52         contact_point = ab * dist
53
54     # get the distance from original point to the closest point
55     square_distance = (contact_point - point).length_squared()
56     return square_distance, contact_point
57
58 # check for floating point equality, may have very similar
value and == will return False
59 def approxEqual(a: float, b: float, epsilon: float=0.0001):
60     return abs(a - b) < epsilon
61
62 # Physics object class (sprite + rigidbody)
63 class Object(pygame.sprite.Sprite, AbstractObject):
64     def __init__(self, position: pm.Vector2, rotation: float,
mass: float, restitution: float, color: tuple[int, int, int],
vertices: list[pm.Vector2]):
65         super().__init__()
66
67         # record object's existence
68         objects.append(self)
69         self.add(objects_group)
70
71         # set a bunch of default values
72         self.position: pm.Vector2 = position
73         self.velocity: pm.Vector2 = pm.Vector2()
74         self.force: pm.Vector2 = pm.Vector2()
75         self.rotation: float = rotation
76         self.angular_velocity: float = 0.0

```

```

77         self.mass: float = mass
78         self.restitution = restitution # object's "bounciness"
79         self.vertices: list[pm.Vector2] = vertices
80         self.translated_vertices: list[pm.Vector2] =
list(vertices)
81         self.forces: list[tuple[pm.Vector2, pm.Vector2]] =
list()
82
83         # get vertices as list of tuple so we can draw it
84         points = list()
85
86         # record max x and max y
87         max_x = 0
88         max_y = 0
89
90         # find the center of mass, assuming constant density
91         centroid_x = 0
92         centroid_y = 0
93         signed_area = 0
94
95         # also find the moment of inertia
96         inertia_x = 0
97         inertia_y = 0
98
99         # iterate over vertices
100        for i in range(len(vertices)):
101            vertex = vertices[i]
102
103            # append to list of points
104            points.append((vertex.x, vertex.y))
105
106            # update maximum x/y
107            max_x = max(vertex.x, max_x)
108            max_y = max(vertex.y, max_y)
109
110            # signed area = .5 * sum(xi * yi+1 - xi+1 * yi)
111            curr_area = vertices[i - 1].x * vertices[i].y -
vertices[i].x * vertices[i - 1].y
112            signed_area += curr_area
113

```

```

114          # centroid = 1/(6A) * sum((x/yi + x/yi+1)(x/yi *
115          #           x/yi+1 - x/yi+1 * x/yi))
116          centroid_x += (vertices[i - 1].x + vertices[i].x) *
curr_area
117          centroid_y += (vertices[i - 1].y + vertices[i].y) *
curr_area
118          # moment of inertia = 1/12 * sum((x/yi + x/yi+1)
(x/yi^2 + x/yi*x/yi+1 + x/yi+1^2))
119          # using gauss shoelace formula
120          inertia_x += curr_area * (vertices[i - 1].x ** 2 +
vertices[i - 1].x * vertices[i].x + vertices[i].x ** 2)
121          inertia_y += curr_area * (vertices[i - 1].y ** 2 +
vertices[i - 1].y * vertices[i].y + vertices[i].y ** 2)
122
123          # finish up some calculations, e.g. CoM = 1/6A * (prev
calculations)
124          signed_area *= 0.5
125
126          density = mass / signed_area
127
128          centroid_x /= 6 * signed_area
129          centroid_y /= 6 * signed_area
130
131          inertia_x /= 12
132          inertia_y /= 12
133          inertia_x -= signed_area * centroid_y ** 2
134          inertia_y -= signed_area * centroid_x ** 2
135
136          # use the values we just found to set the center of
mass and moment of inertia
137          self.mass_center: pm.Vector2 = pm.Vector2(centroid_x,
centroid_y)
138          self.inertia = (inertia_x + inertia_y) * density
139
140          # minimum size of pygame surface to fit the shape onto
141          width = max_x
142          height = max_y
143
144          # create transparent surface

```

```

145         self.surface = pygame.Surface((width, height),
pygame.SRCALPHA, 32)
146         self.image = self.surface
147
148         # draw polygon onto it
149         pygame.draw.polygon(self.image, color, points)
150
151         # get bounding rectangle of image
152         self.rect = self.image.get_rect(center=self.position)
153
154     def step(self, timestep):
155         # apply all forces that occurred since last update
156         for force, point in self.forces:
157             self.__applyForce(force, point, timestep)
158         self.forces.clear()
159
160         # update position and rotation based on velocities
161         self.position += self.velocity * timestep
162         self.rotation += self.angular_velocity * timestep
163
164         # update the positions of each vertex based on new
position and rotation
165         for i in range(len(self.vertices)):
166             # make a copy of vertex so we don't change original
167             vertex = pm.Vector2()
168             vertex.x = self.vertices[i].x
169             vertex.y = self.vertices[i].y
170
171             # center vertices around (0, 0)
172             vertex -= self.mass_center
173
174             # using formula to rotate points
175             x = vertex.x * math.cos(self.rotation) - vertex.y *
math.sin(self.rotation)
176             y = vertex.y * math.cos(self.rotation) + vertex.x *
math.sin(self.rotation)
177             vertex.x = x
178             vertex.y = y
179
180             # translate to world position and save new vertex

```

```

181         vertex += self.position
182         self.translated_vertices[i] = vertex
183
184     def update(self, *args, **kwargs):
185         # redraw polygon image
186         self.image = pygame.transform.rotate(self.surface,
187         math.degrees(-self.rotation))
188
189         # update bounding box position
190         self.rect = self.image.get_rect(center=self.position)
191
192     def __applyForce(self, force: pm.Vector2, point:
193         pm.Vector2, timestep: float):
194         # F = ma
195         # a = F/m
196         # v = at = (F/m)t
197         self.velocity += force * timestep * (1.0 / self.mass)
198
199         # torque = cross product of force and displacement
200         torque = point.x * force.y - point.y * force.x
201
202         # angular acceleration = torque / moment of inertia
203         # angular velocity = angular acceleration * time
204         self.angular_velocity += (torque / self.inertia) *
205         timestep
206
207     def applyForce(self, force: pm.Vector2, point: pm.Vector2):
208         self.forces.append((force, point))
209
210     def applyImpulse(self, force: pm.Vector2, point:
211         pm.Vector2):
212         # impulse = instantaneous force (apply the whole force
213         # instantly, not over time)
214         self.__applyForce(force, point, 1.0)
215
216     def getAxes(self):
217         # get axes for SAT collisions
218         # an axis is just a perpendicular vector from each face
219         axes: list[pm.Vector2] = list()
220         for i in range(len(self.translated_vertices)):

```

```

216         # iterate vertices 2 at a time, gives faces
217         edge = self.translated_vertices[i] -
218             self.translated_vertices[i - 1]
219
220         # rotate face vector and normalise to get axis
221         axes.append(pm.Vector2(-edge.y,
222             edge.x).normalize())
223         return axes
224
225     def project(self, axis):
226         # project each face onto an axis, and get minimum and
227         # maximum displacements
228         # pick a projection to start with (vertex 0)
229         # projection is found using dot product
230         min_p = axis.dot(self.translated_vertices[0])
231         max_p = min_p
232
233         # iterate over the remaining vertices
234         for i in range(1, len(self.translated_vertices)):
235             # project each one, and store new minimum and
236             # maximum projection
237             proj = axis.dot(self.translated_vertices[i])
238             min_p = min(proj, min_p)
239             max_p = max(proj, max_p)
240
241         return pm.Vector2(min_p, max_p)
242
243     def findContactPoints(self, other):
244         # find the points of contact between two polygons
245         # maximum of two contact points, currently we dont know
246         # any (0 points right now)
247         point_a = pm.Vector2()
248         point_b = pm.Vector2()
249         num_points = 0
250
251         # store minimum distance, we don't know right now, so
252         # just assume the worst (infinite)
253         min_distance = float("inf")
254
255         # iterate over vertices of object
256         for i in range(len(self.translated_vertices)):

```

```

250     point = self.translated_vertices[i]
251
252         # iterate over other object
253         for j in range(len(other.translated_vertices)):
254             # get a face of the other object
255             vertex_a = other.translated_vertices[j - 1]
256             vertex_b = other.translated_vertices[j]
257
258             # compute closest point of contact between this
shape's point and other shape's face
259             distance, contact_point =
nearestPoint(vertex_a, vertex_b, point)
260
261             # if we already have a similar distance,
possibly another contact point
262             if approxEqual(distance, min_distance):
263                 # make sure it isn't just the same point
found in reverse
264                 if not approxEqual(contact_point.x,
point_a.x) and approxEqual(contact_point.y, point_a.y):
265                     # if a different point, then this is
the second contact point
266                     num_points = 2
267                     point_b = contact_point
268                     # if this is a new smallest distance, this must
be the contact point
269             elif distance < min_distance:
270                 min_distance = distance
271                 num_points = 1
272                 point_a = contact_point
273
274             # repeat this process, check the other shape's points
over this shape's faces
275             for i in range(len(other.translated_vertices)):
276                 point = other.translated_vertices[i]
277                 for j in range(len(self.translated_vertices)):
278                     vertex_a = self.translated_vertices[j - 1]
279                     vertex_b = self.translated_vertices[j]
280                     distance, contact_point =
nearestPoint(vertex_a, vertex_b, point)

```

```

281
282             if approxEqual(distance, min_distance):
283                 if not approxEqual(contact_point.x,
284 point_a.x) and approxEqual(contact_point.y, point_a.y):
285                     num_points = 2
286                     point_b = contact_point
287                 elif distance < min_distance:
288                     min_distance = distance
289                     num_points = 1
290                     point_a = contact_point
291
292         # return a list of contact points, based on number of
293         # contact points
294         if num_points == 1:
295             return [point_a]
296         elif num_points == 2:
297             return [point_a, point_b]
298         else:
299             return None
300
301     def getCollisionNormal(self, other):
302         # find the collision normal
303         # firstly get a list of all axes of both shapes
304         axes = self.getAxes()
305         axes.extend(other.getAxes())
306
307         # some variables to track inside the loop
308         overlap = float("inf")
309         closest_axis = None
310         for axis in axes:
311             # project each shape along an axis
312             proj_s = self.project(axis)
313             proj_o = other.project(axis)
314
315             # check if the projections overlap
316             if not ((proj_s.x < proj_o.x < proj_s.y) or
317 (proj_o.x < proj_s.x < proj_o.y)):
318                 # if there is no overlap, there is no collision
319                 return None
320             else:

```

```

318             # if there is an overlap, compute the size of
the overlap
319             cur_overlap = min(proj_s.y, proj_o.y) -
max(proj_s.x, proj_o.x)
320             # if overlap is smaller, this is a closer axis,
so update values
321             if cur_overlap < overlap:
322                 overlap = cur_overlap
323                 closest_axis = axis
324
325             return closest_axis
326
327     def collide(self, other):
328         # find the collision normal, if no normal then there's
no collision
329         normal = self.getCollisionNormal(other)
330         if normal is None: return
331
332         # find contact points, if there are no contact points
then there's no collision
333         contact_points = self.findContactPoints(other)
334         if contact_points is None: return
335
336         # assume restitution to be the minimum of both objects
337         restitution = min(self.restitution, other.restitution)
338
339         impulses = []
340         r_as = []
341         r_bs = []
342
343         # loop over the contact points
344         for point in contact_points:
345             # calculate the vectors from contact point to
centers of mass
346             r_a = point - self.position
347             r_b = point - other.position
348
349             # store the vectors
350             r_as.append(r_a)
351             r_bs.append(r_b)

```

```

352
353             # calculate perpendicular vectors (used in angular
354             # calculations)
354             rp_a = pm.Vector2(-r_a.y, r_a.x)
355             rp_b = pm.Vector2(-r_b.y, r_b.x)
356
357             # relative velocity = linear velocity + angular
358             # velocity difference of both objects
358             a_velocity = rp_a * self.angular_velocity +
359             self.velocity
359             b_velocity = rp_b * other.angular_velocity +
360             other.velocity
360
361             relative_velocity = b_velocity - a_velocity
362
363             # contact speed is the relative velocity in the
364             # direction of the collision
364             contact_speed = relative_velocity.dot(normal)
365
366             # if contact speed is positive, objects are already
367             # moving away from each other
367             if contact_speed > 0.0:
368                 impulses.append(None)
369                 continue
370
371             # calculate inverse mass
372             inv_mass_a = 1.0 / self.mass
373             inv_mass_b = 1.0 / other.mass
374
375             # adjustment values to account for rotational
376             # velocities
376             r_adj_a = (rp_a.dot(normal)) ** 2 / self.inertia
377             r_adj_b = (rp_b.dot(normal)) ** 2 / other.inertia
378
379             # calculate impulse J using formula
380             j = (-(1.0 + restitution) * contact_speed) /
381             (inv_mass_a + inv_mass_b + r_adj_a + r_adj_b)
381
382             # share J between contact points
382             j /= len(contact_points)

```

```

384
385         # apply impulse along direction of collision
386         impulse = normal * j
387
388         # store the impulse
389         impulses.append(impulse)
390
391     for i in range(len(contact_points)):
392         impulse = impulses[i]
393         if impulse is None:
394             # None impulse means objects were moving away
395             continue
396
397         # apply impulse to objects, in opposite directions
398         self.applyImpulse(-impulse, r_as[i])
399         other.applyImpulse(impulse, r_bs[i])
400
401
402 def main():
403     # pygame shenanigans
404     pygame.init()
405     screen = pygame.display.set_mode((800, 600))
406     clock = pygame.time.Clock()
407     running = True
408
409     # points to describe a pentagon
410     pentagon = [
411         pm.Vector2(100, 0),
412         pm.Vector2(200, 80),
413         pm.Vector2(140, 200),
414         pm.Vector2(60, 200),
415         pm.Vector2(0, 80)
416     ]
417
418     # points to describe a long stick
419     stick = [
420         pm.Vector2(0, 0),
421         pm.Vector2(50, 0),
422         pm.Vector2(50, 300),
423         pm.Vector2(0, 300)

```

```

424     ]
425
426     # create a rectangular object
427     object_1 = Object(pm.Vector2(0, 0), 0, 5, 0.5, (255, 50,
50), stick)
428     object_1.update()
429
430     # create a pentagonal object
431     object_2 = Object(pm.Vector2(300, 300), 0, 1, 0.5, (255,
50, 50), pentagon)
432
433     # apply some acceleration to the objects
434     object_1.applyImpulse(pm.Vector2(250, 250), pm.Vector2(200,
0))
435     object_2.applyImpulse(pm.Vector2(-10, -10), pm.Vector2(0,
0))
436
437     while running:
438         # pygame updates
439         for event in pygame.event.get():
440             if event.type == pygame.QUIT:
441                 running = False
442
443         # clear screen
444         screen.fill((255, 255, 255))
445
446         # update and draw objects
447         objects_group.update()
448         objects_group.draw(screen)
449
450         # check collisions
451         object_2.collide(object_1)
452
453         # update physics
454         physics()
455
456         # pygame stuff
457         pygame.display.flip()
458         clock.tick(60)
459

```

```
460 if __name__ == '__main__':
461     main()
```

Three Dimensional Graphics

Additionally, I wanted to model how a 3D physics simulator may look to the final user, and so I created a simple 3D renderer using OpenGL, and implemented a basic scenario of a cube hitting against a solid wall and bouncing off it. This code will be useful to the final application, as it will become the basis of my rendering engine. I will also need the ability to see the objects I am simulating while testing the physics simulator, as it may be hard to tell whether an object is moving normally from looking at a table of data. This prototype focuses mostly on the graphical aspects, the collision is not simulated, but is instead animated.

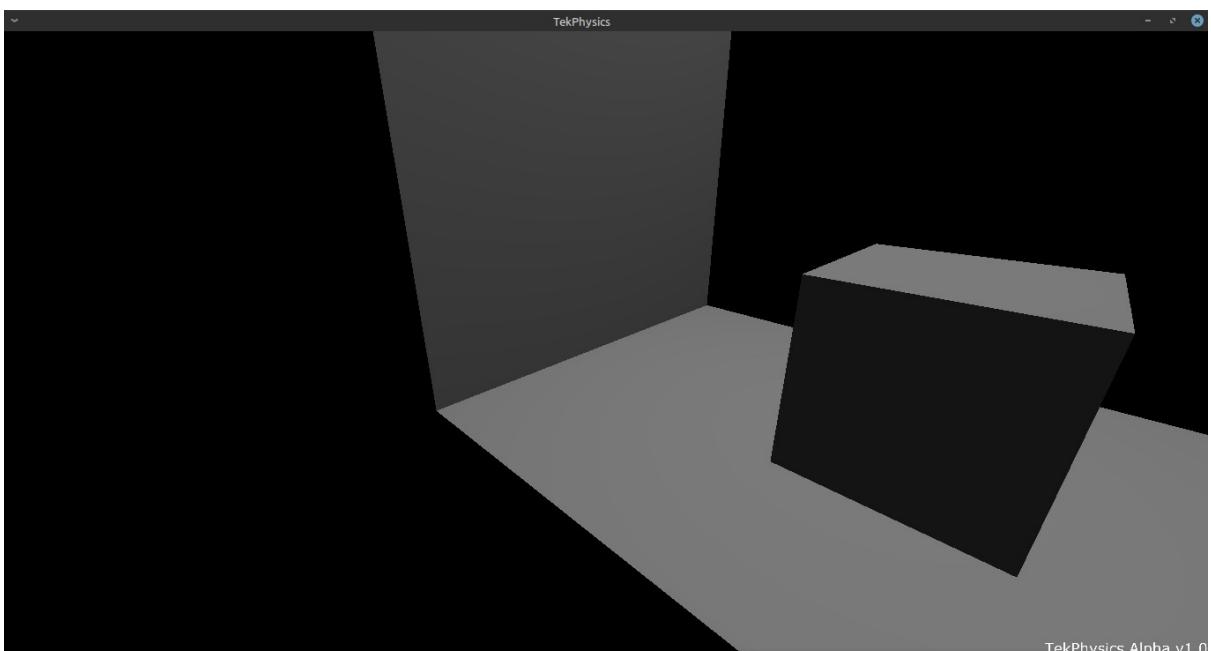


Figure 18: A screenshot of the prototype working

I have also created a short video of the prototype working, which is available on YouTube via this link:

<https://www.youtube.com/watch?v=wPNemj0AsCE>

Or via this QR Code:



Below I have included some of the source code that was used to create the animated scene. However, most of the code for rendering the scene such as shaders, meshes, and text rendering were eventually used in the final code, hence I have decided to omit them from this section as they are also included in later parts of this document.

```
1 #include <stdio.h>
2 #include <glad/glad.h>
3 #include <cglm/mat4.h>
4 #include "core/exception.h"
5 #include "tekgl/mesh.h"
6 #include "tekgl/shader.h"
7 #include "tekgl/font.h"
8 #include "tekgl/text.h"
9
10 #include <cglm/cam.h>
11
12 #include "tekgl/manager.h"
13 #include "tekgui/primitives.h"
14
15 #include <time.h>
16
17 #define printException(x) tekLog(x)
18
19 float width, height;
20 mat4 perp_projection;
21
22 void tekMainFramebufferCallback(const int fb_width, const int
fb_height) {
23     width = (float)fb_width;
24     height = (float)fb_height;
25     glm_perspective(1.2f, width / height, 0.1f, 100.0f,
perp_projection);
```

```

26 }
27
28 int render() {
29     // create the main window
30     tekChainThrow(tekInit("TekPhysics", 640, 480));
31     width = 640.0f;
32     height = 480.0f;
33
34     // add a framebuffer callback to allow for resizing the
window
35     tekAddFramebufferCallback(tekMainFramebufferCallback);
36
37     // create a font to render the program name + version
38     TekBitmapFont verdana;
39     printException(tekCreateFreeType());
40
41     tekChainThrow(tekCreateBitmapFont("../res/verdana.ttf", 0,
64, &verdana));
42
43     TekText text;
44     tekChainThrow(tekCreateText("TekPhysics Alpha v1.0", 20,
&verdana, &text));
45
46     // once font has been used, we can delete the font creator
47     tekDeleteFreeType();
48
49     // some OpenGL stuff for blending and shading etc.
50     glEnable(GL_DEPTH_TEST);
51     glDepthFunc(GL_LESS);
52
53     glEnable(GL_BLEND);
54     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
55
56     // the mesh data for the walls/floor
57     TekMesh walls = {};
58
59     const float wall_vertices[] = {
60         0.0f, 6.0f, -3.0f, 1.0f, 0.0f, 0.0f,
61         0.0f, 6.0f, 3.0f, 1.0f, 0.0f, 0.0f,
62         0.0f, 0.0f, 3.0f, 1.0f, 0.0f, 0.0f,

```

```

63         0.0f, 0.0f, -3.0f, 1.0f, 0.0f, 0.0f,
64         0.0f, 0.0f, -3.0f, 0.0f, 1.0f, 0.0f,
65         0.0f, 0.0f, 3.0f, 0.0f, 1.0f, 0.0f,
66         12.0f, 0.0f, 3.0f, 0.0f, 1.0f, 0.0f,
67         12.0f, 0.0f, -3.0f, 0.0f, 1.0f, 0.0f
68     };
69
70     const uint wall_indices[] = {
71         0, 1, 2,
72         0, 2, 3,
73         4, 5, 6,
74         4, 6, 7
75     };
76
77     const uint wall_layout[] = {
78         3, 3
79     };
80
81     tekChainThrow(tekCreateMesh(wall_vertices, 48,
wall_indices, 12, wall_layout, 2, &walls));
82
83     // mesh data for the cube (well, half of a cube that's
visible)
84     TekMesh cube = {};
85
86     const float cube_vertices[] = {
87         -1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
88         -1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f,
89         1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f,
90         1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
91         1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f,
92         1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 0.0f,
93         1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 0.0f,
94         1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f,
95         -1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
96         1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
97         1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
98         -1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f
99     };
100

```

```

101     const uint cube_indices[] = {
102         0, 1, 2,
103         0, 2, 3,
104         4, 6, 5,
105         4, 7, 6,
106         8, 10, 9,
107         8, 11, 10
108     };
109
110     const uint cube_layout[] = {
111         3, 3
112     };
113
114     tekCreateMesh(cube_vertices, 72, cube_indices, 36,
cube_layout, 2, &cube);
115
116     // set up camera
117     vec3 camera_position = {5.0f, 3.0f, 5.0f};
118     vec3 light_color = {0.4f, 0.4f, 0.4f};
119     vec3 light_position = {4.0f, 12.0f, 0.0f};
120
121     // set up the shaders
122     uint shader_program;
123
tekChainThrow(tekCreateShaderProgramVF("../shader/vertex.glvs",
"../shader/fragment.glfs", &shader_program));
124
125     tekBindShaderProgram(shader_program);
126
127     tekShaderUniformVec3(shader_program, "light_color",
light_color);
128     tekShaderUniformVec3(shader_program, "light_position",
light_position);
129     tekShaderUniformVec3(shader_program, "camera_pos",
camera_position);
130
131     mat4 wall_model;
132     mat4 view;
133
134     glm_mat4_identity(wall_model);

```

```

135
136     vec3 center = {0.0f, 0.0f, 0.0f};
137     vec3 up = {0.0f, 1.0f, 0.0f};
138     glm_lookat(camera_position, center, up, view);
139
140     // creating the projection matrix for the shader
141     glm_perspective(1.2f, width / height, 0.1f, 100.0f,
perp_projection);
142
143     tekShaderUniformMat4(shader_program, "view", view);
144
145     vec3 cube_position = {3.0f, 1.0f, 0.0f};
146     mat4 cube_model;
147
148     // keeping track of time to maintain cube moving at
constant speed.
149     struct timespec t;
150     clock_gettime(CLOCK_REALTIME, &t);
151     long time_ns = t.tv_sec * 1000000000 + t.tv_nsec;
152     double delta = 0.0;
153     double multiplier = -1.0;
154
155     while (tekRunning()) {
156         clock_gettime(CLOCK_REALTIME, &t);
157         long delta_ns = (t.tv_sec * 1000000000 + t.tv_nsec) -
time_ns;
158         delta = ((double)delta_ns) / 1000000000.0;
159         time_ns = t.tv_sec * 1000000000 + t.tv_nsec;
160
161         // clear the background colour
162         glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
163
164         // update matrices to cause movement of cube
165         tekBindShaderProgram(shader_program);
166         tekShaderUniformMat4(shader_program, "projection",
perp_projection);
167
168         tekShaderUniformMat4(shader_program, "model",
wall_model);
169         tekDrawMesh(&walls);

```

```

170
171         // move cube forward depending on whether or not it has
already "hit the wall"
172         cube_position[0] += delta * multiplier;
173         if (cube_position[0] < 1.0f) {
174             cube_position[0] = 1.0f;
175             multiplier = 1.0;
176         }
177
178         // update model matrix
179         glm_translate_make(cube_model, cube_position);
180         tekShaderUniformMat4(shader_program, "model",
cube_model);
181         tekDrawMesh(&cube);
182
183         // draw version text
184         printException(tekDrawText(&text, width - 185.0f,
5.0f));
185         tekUpdate();
186     }
187
188     // clean up
189     tekDeleteShaderProgram(shader_program);
190     tekDeleteMesh(&walls);
191     tekDeleteTextEngine();
192     tekDeleteText(&text);
193     tekDelete();
194     return SUCCESS;
195 }
196
197 int main(void) {
198     tekInitExceptions();
199     tekLog(render());
200     tekCloseExceptions();
201 }
```

Design

Mission Statement

TekPhysics will be an educational desktop application that is capable of performing 3D rigid body physics simulations. It will be used mostly by teachers, who can display the program on a projector or large screen to show in a classroom. The program will provide a graphical user interface to allow the user to edit different objects in a scenario before running, allowing them to simulate a variety of different problems. The program will also output displacement or velocity time graphs in visual and numerical form, to allow for further use in class.

The complexity of this project is with rendering in three dimensions, and also with checking and processing collisions between objects in three dimensions. Both of these areas will require large optimisations to the code, as they are computationally expensive tasks that, if not optimised, will not allow the program to run in real-time.

Project Overview

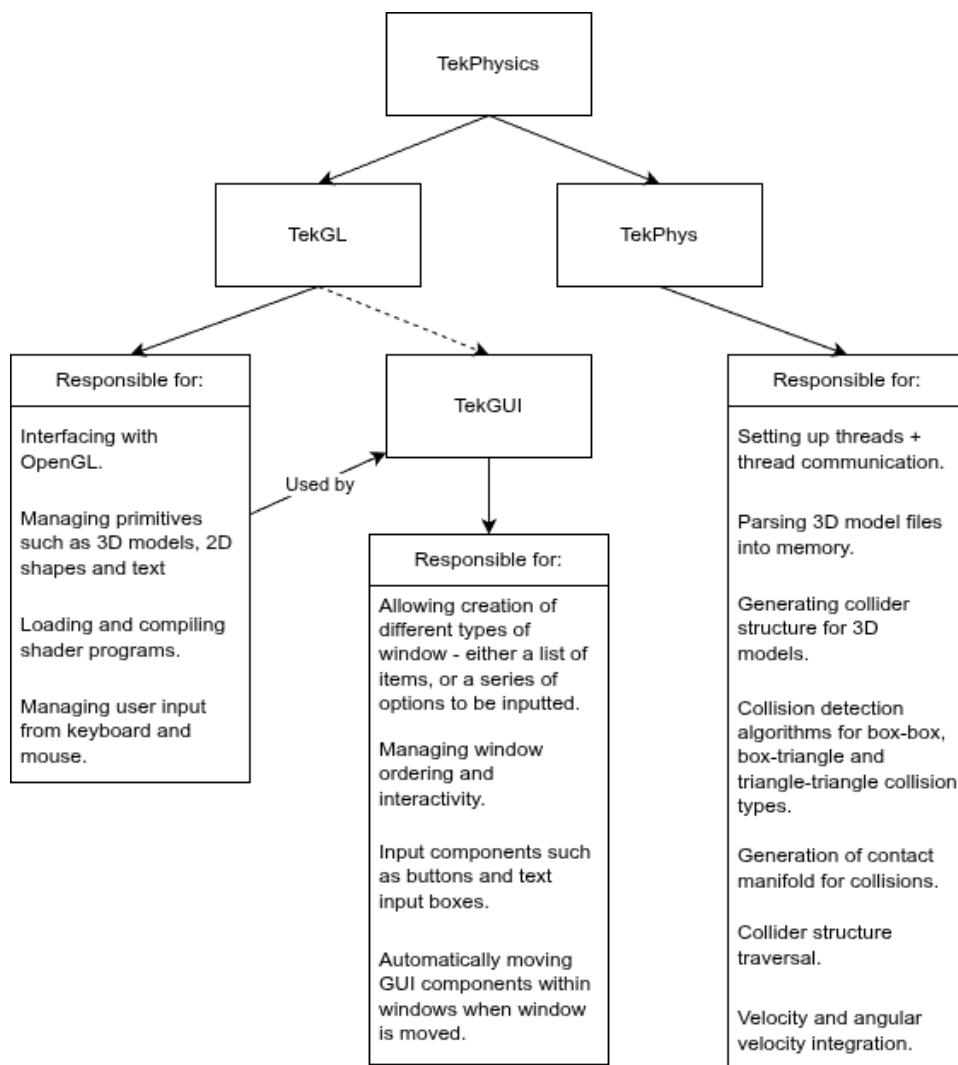


Figure 19: A diagram showing the general project structure of TekPhysics

User Interface

The user interface is a very important aspect of this project. It is intended as a program to help explain complicated aspects of mechanics to students in the classroom, so it almost certainly should have a graphical user interface rather than a text-based interface. This program would not be particularly useful if all it outputted was a set of coordinates and velocities for two objects, because it wouldn't help people to get an intuitive understanding of how these problems would look if they happened in reality. My end user also specifically requested that the user interface was "Graphical".

User Interface Layout

The user interface should contain a series of menus that can be accessed in a linear progression. Below is a diagram to represent how the different menus are linked to each other:

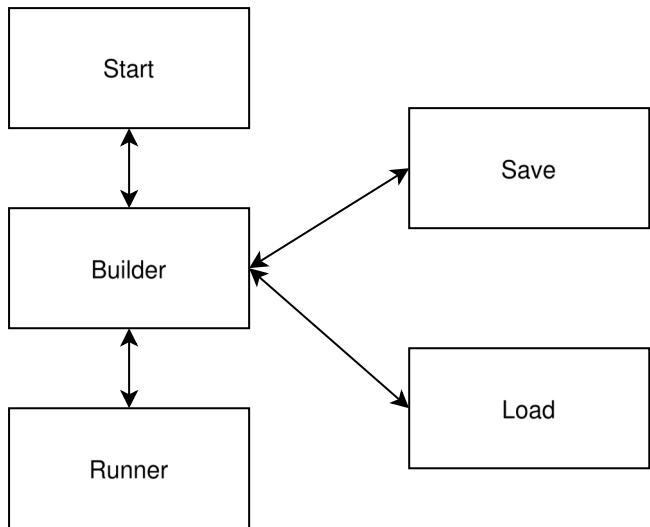


Figure 20: A diagram explaining the flow of the GUI.

- The start menu should be a simple page that simply displays a logo and a start button.
- The builder menu should allow the user to build a scenario by adding and editing different objects. It should also allow some scenario options to be edited such as the acceleration due to gravity.
- The save menu should give a user the option to enter a name of a file to save their scenario into.
- The load menu should give a user the option to load a file containing a scenario they have previously created.
- The runner menu should contain options that are useful while the simulation is in progress such as the simulation speed and a pause/play button etc.

Main Menu

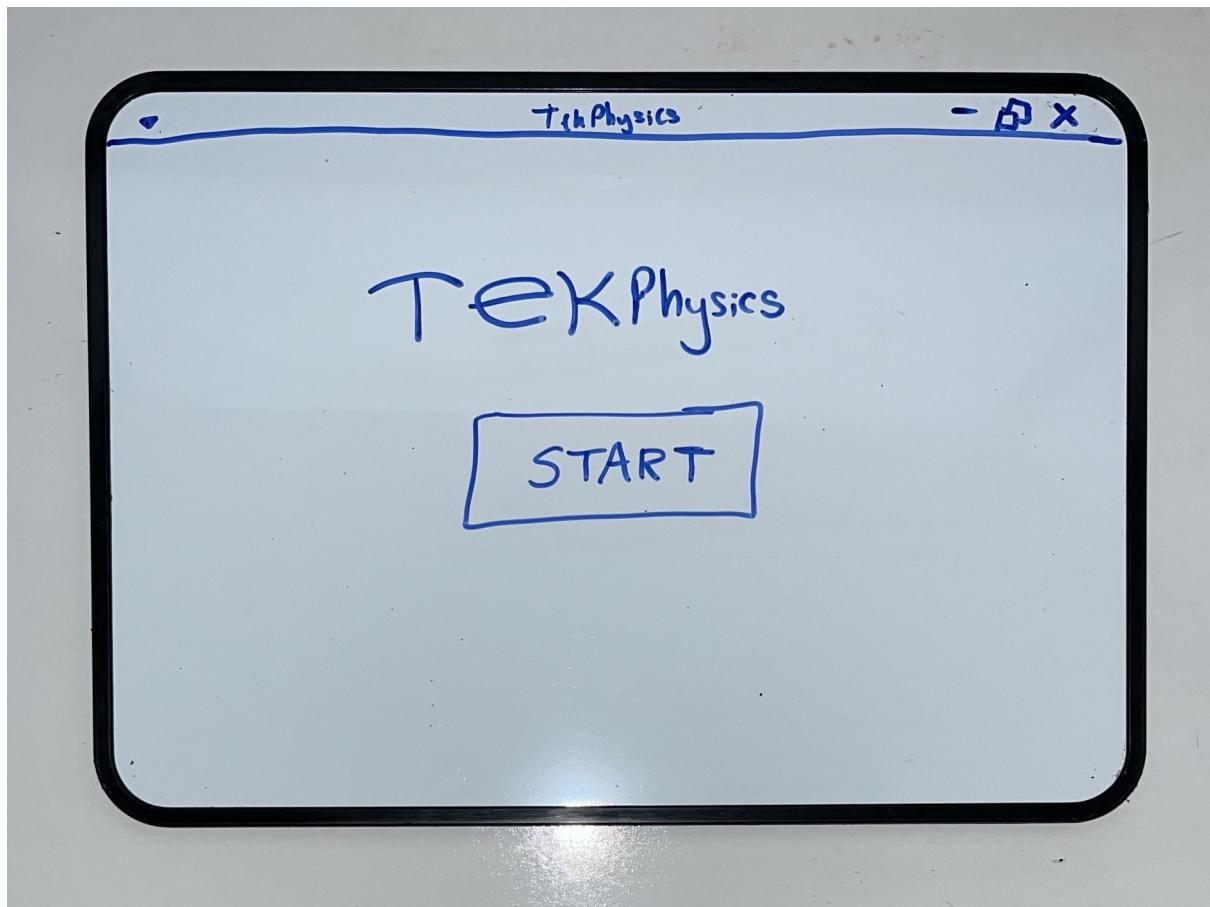


Figure 21: A drawing of how the main menu should look.

The main menu should look like this. It should simply display the logo and a start button in the middle of the screen.

Builder Menu

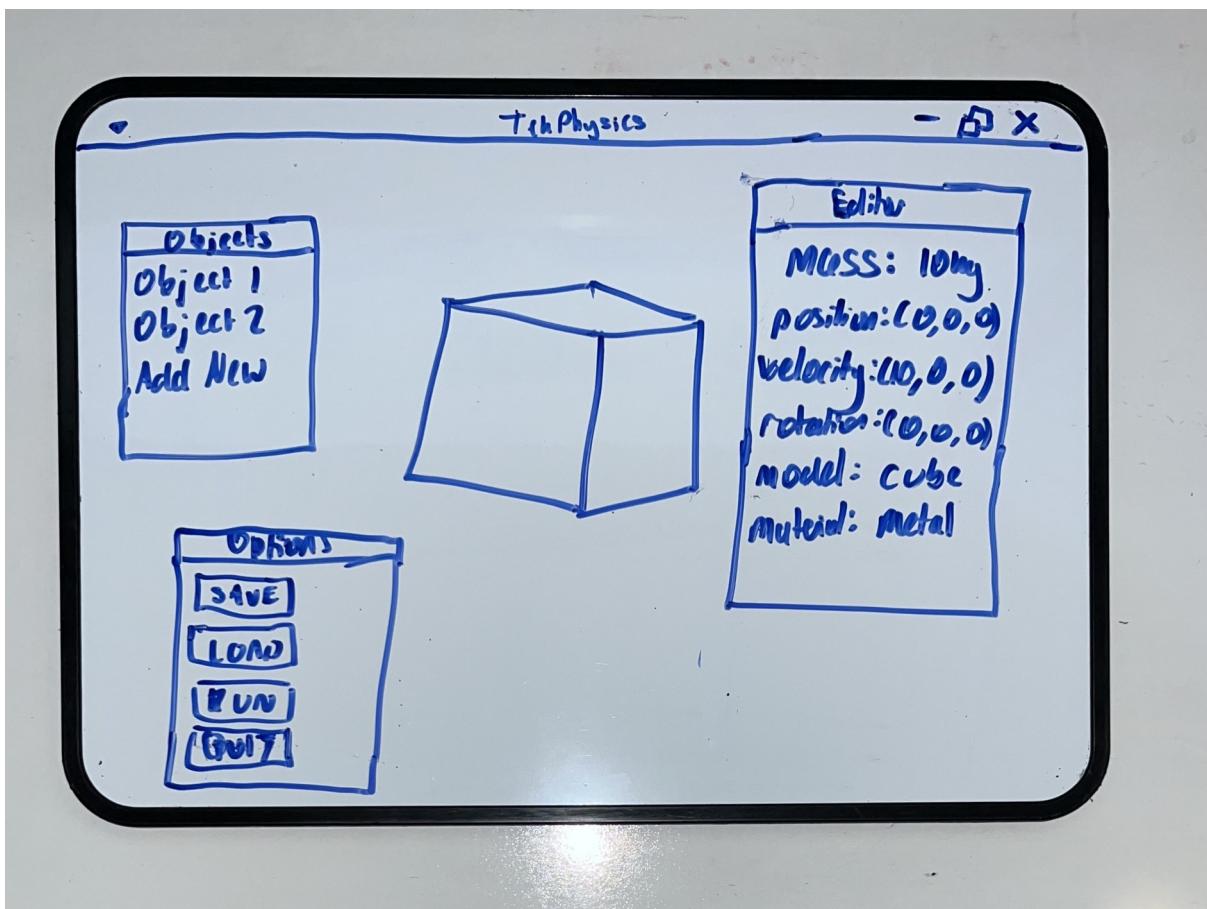


Figure 22: A drawing of the builder / scenario editor menu.

The builder menu should have a similar layout to this. There should be a window that displays all the objects in the scene as a list. There should also be a window that allows a selected object to be edited, with options to edit the mass, position, rotation, velocity and other properties. Finally, there should be a window to select different options, such as saving the scenario, loading a new scenario, running the current scenario and quitting to the main menu. In this menu, the objects in the scenario should also be rendered, but unable to move. This allows the user to see any changes they make to the scenario before they press run.

Runner Menu

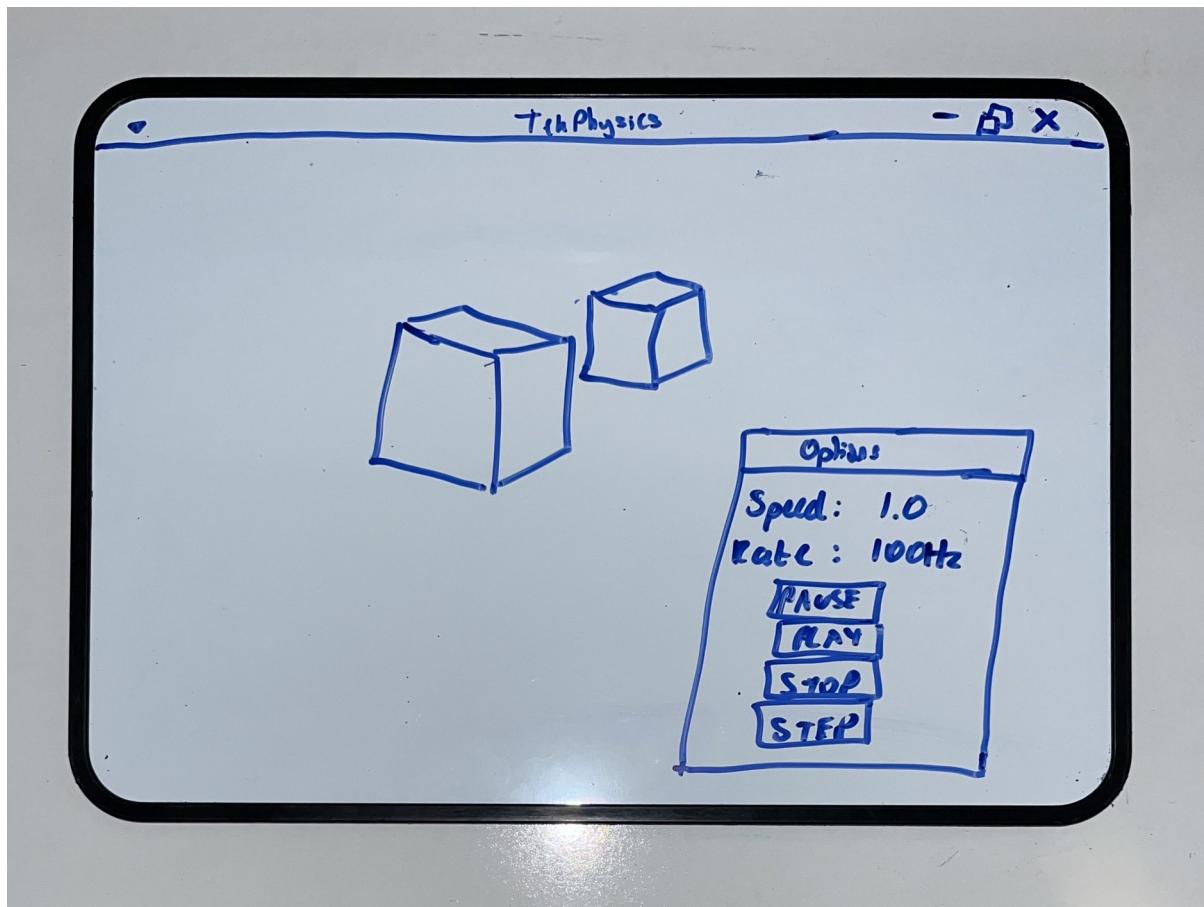


Figure 23: A drawing of the runner menu.

The runner menu should be primarily for showing how the simulation progresses over time, so there should be a minimal amount of windows on the screen. There should be a window that allows settings of the simulation to be changed, as well as being able to play, pause, stop and step through the simulation. There should also be a window or text that displays data about a requested object. This can be used if a visual representation is less helpful than a numerical one, for example if explaining how a coefficient of restitution affects the final velocity of a collision.

Save Menu and Load Menu

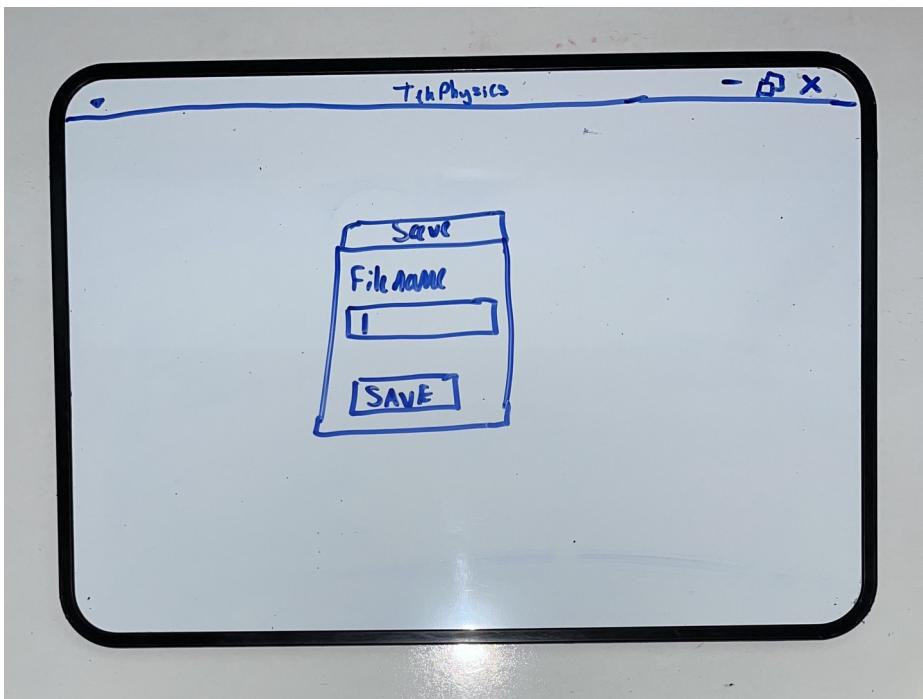


Figure 24: A drawing of the save menu.

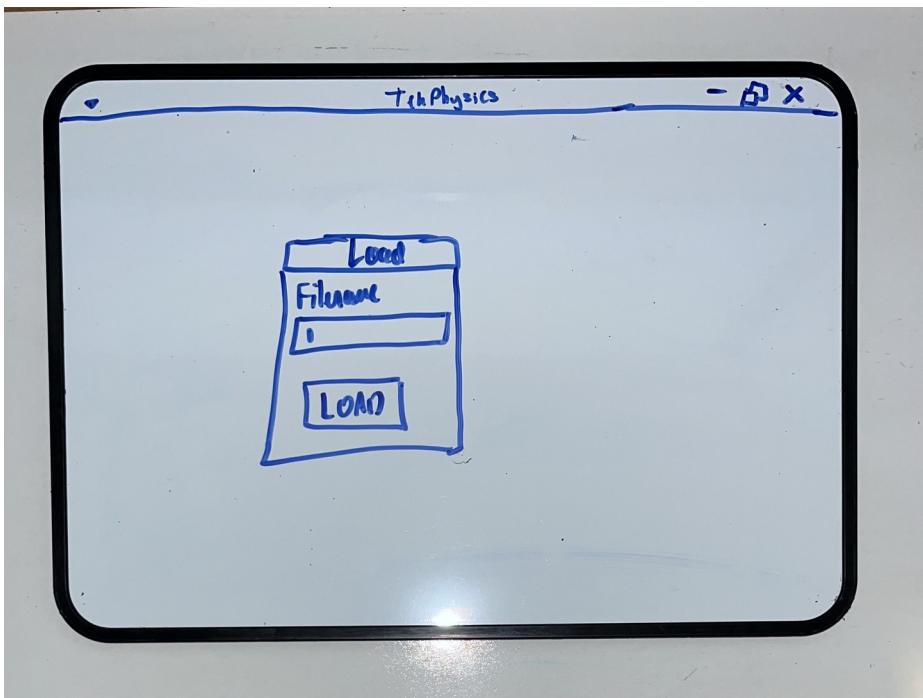


Figure 25: A drawing of the load menu.

The save and load menu should both be similar, with the option to enter a filename and then either save to or load from that file. The load menu will need to check that the specified file exists before allowing the user to continue with loading it.

Required Libraries

In order to function correctly, TekPhysics will rely on some external libraries. These are used for a variety of different reasons as detailed below:

1. GLFW – Provides API for creating windows and receiving input events.
2. GLAD – Manages the function pointers of OpenGL so that it can be interfaced by the program.
3. stb – Allows for images such as .png and .jpeg to be decompressed and converted to a bitmap image for use as textures.
4. FreeType – Allows for true type font files to be read, so each glyph can be read and used when rendering text.
5. LAPACK – Linear Algebra Package, has a variety of functions but will only be used to find eigenvectors and eigenvalues of a matrix.

Physics vs Graphics

In order to keep the project maintainable, it should be divided into two main sections – physics and graphics. The physics side of things should remain completely separated from the graphics side where possible, apart from when the user interacts with a graphical interface to edit the simulation. This division could cause some confusion, as the project will contain two data structures that appear to be quite similar, a “Body” and an “Entity”. However, it will be much better in the long run.

A “body” is a purely physics-based construct, it will contain the data needed to simulate an object such as position, velocity, rotation, angular velocity, mass, coefficients of friction and restitution, a collider structure, and some other properties such as if the body is immovable, and the scale of the body.

An “entity” will also contain the position and rotation of the object. However, it will not record the velocity, mass or any other properties. This is because an entity is a representation of the object purely for graphical purposes, and so it contains a rendering material and a 3D mesh to draw the object with. This abstraction from the actual body on the physics side means that it is impossible to accidentally change a value that could lead to an unexpected bug. However, to use the physics engine we need to be able to create or change the bodies in some way.

This is why there will also be a third data structure, a “Body Snapshot”. This is a local, abstracted copy of a body on the physics side, containing only a position, rotation, velocity, mass, friction, restitution, and filenames of the 3D model and material to use. You can imagine it as a descriptor of the real body that the physics and graphics sides use to communicate. Changing a body snapshot will have no effect on the real body unless that body snapshot is explicitly sent to the engine, and likewise for the graphics side.

The graphics side of the program is handled by the areas “TekGL” and “TekGui”, while the physics side of the program is handled by “TekPhys”.

TekPhys

“TekPhys” is the part of TekPhysics that will handle all of the complicated mathematical parts of the program. TekPhys is designed to run independently of the rest of the program and only be interfaced through an event queue, this would allow for the program to be updated later down the line if a different rendering system was to be implemented.

There are several key components that need to be implemented in this section.

Threading and Thread Communication

TekPhys will communicate to the rest of the program using two thread-safe queues. These are:

- The “Event Queue”
 - This will essentially be the queue that handles input from the rest of the program.
 - For example, there could be a “change simulation speed” event, a “stop simulation” event.
 - TekPhys will only dequeue from this queue, and the main thread of the program will only enqueue to this queue.
- The “State Queue”
 - This will be the queue that handles the output of the physics simulation.
 - For example, there could be an “update entity position” event, or perhaps an “exception encountered” event.
 - TekPhys will only enqueue to this queue, and the main thread will only dequeue from this queue.

By designating each thread a queue to enqueue to, and a queue to dequeue from, a lot of the race conditions and other troubles of multithreading are avoided. It also allows for a much simpler implementation of a multithreaded queue – a circular queue where the front and read pointers are now atomic variables to avoid both threads changing them at once.

Each queue will have a specific data structure that represents a possible event. These are appropriately named an “Event” and a “State” respectively. Both data structures are required to contain a “type”. This will be an integer that describes the type of the event. A state is also required to contain an “object ID”, as the vast majority of states are related to an object anyway, so it’s sensible to have this option by default.

The possible types for an event should be as follows:

1. Quit – stop the program
2. Change mode – change the mode between building and running etc.
3. Create body – create a new body
4. Delete body – delete a body
5. Update body – update a body with new position etc.
6. Clear – delete all bodies
7. Time – update the rate of passage of time, and time stepping intervals
8. Pause – stop the simulation temporarily
9. Step – allow a single simulation step to occur before pausing again
10. Gravity – update the acceleration downwards due to gravity

Furthermore, the types for a state should be:

1. Message – display a message from the thread
2. Exception – return an exception from the thread
3. Create entity – create a new entity to represent a body
4. Delete entity – delete an entity that represented a body
5. Update entity – update an entity with new position and rotation

Each event and state should also have the possibility to contain data related to its type. These pieces of data are not required all at once – each event/state contains only one of these at a time, and each one relates to a different type. For an event this includes only one of:

- A pair of “body snapshot” data and body ID number. This is so that for the body creation, deletion and update types, TekPhys knows what to update and what to update it with. On deletion, the body data can be left blank.
- An integer that states the mode of the program. This allows the physics engine to react appropriately to switching from running mode to building mode and vice-versa.
- A pair of numbers that contain the rate of passage of time and the time period between each update. This allows the timing of the engine to be adjusted.
- A boolean value that stores if the engine should be paused or not.
- A number that stores the acceleration due to gravity.

For a state this includes only one of:

- A string that contains a message to be sent to the main thread.

- An integer that records an exception code if there was an exception.
- A collection of data related to the creation of a new entity to represent a body. This includes:
 - A filename from which to read a 3D model.
 - A filename from which to read a material for this object.
 - A coordinate for the centre position.
 - A quaternion for the rotation.
 - A vector containing the scale of the object.
- A collection of data to update an existing entity, this is similar to creation, but only including the position, rotation and scale.

Physics Engine Loop

The primary goal of a physics engine is to simulate the movement of objects over time. In reality, time progresses in a continuous manner at a constant rate. Computers cannot calculate the position of each object at every possible moment in time because there are an infinite number of possible moments that could be calculated. Instead, the physics engine will approximate the progression of time using discrete time intervals, also known as a “time step”. So instead of calculating at every moment, we could calculate once every 0.05 seconds instead, and estimate what would have happened to the object during the time that has passed since the last update.

To do this, we can create an “engine loop”, which is a piece of code that attempts to repeat continuously until the program ends, waiting for the time interval to pass before each iteration. The loop has three key parts:

1. Receive all messages from the event queue
 - Enter an inner loop that dequeues the event queue until it is empty.
 - For each event that is dequeued, retrieve the event type.
 - Depending on the event type, interpret the event data as needed, for example if a body creation event type was received, interpret the event data as a body snapshot and body id, and update the body with the new data.
2. Process collisions
 - Enter a nested for loop, where the outer loop repeats for the number of bodies, and the inner loop repeats for one less than the current index of the outer loop. This should have the effect of finding all combinations of bodies without any repeats when ignoring the order of the pair.
 - Any pair of bodies where both bodies are immovable can be skipped.

- For each pair of bodies, decide whether their colliders are intersecting, and if so, retrieve the points of contact, depth of contact points, contact normals, and tangents.
- Add any contact information along with the associated bodies to an array.
- For each item of this array, calculate some other properties such as the vector between centres, the difference in velocities, and a Baumgarte term (velocity correction term).
- Now enter the iterative solver loop
 - . For each body, create a set of constraints for the normal and tangent directions.
 - . A single solver step will update the velocity of the object to best conform to this constraint.
 - . The velocity of the object is immediately updated.
- The iterative solver loop should repeat for a set number of times, for example 10 or 20 iterations. This allows changes in velocity to propagate through objects that are not directly in contact, for example three spheres in a row – the first sphere could push the second sphere into the third sphere even though the first and the third are not in contact. This solver loop means that the solution of the velocity constraints can reach a global solution that satisfies all of the bodies.

3. Update bodies

- For each body, integrate the current velocity and angular velocity with respect to time, over the duration of this physics step.
- You will also need to account for acceleration due to gravity in this step, by multiplying the acceleration due to gravity by the time step and adding it to the velocity of the object before integrating.
- Add the change in position and rotation to the current position and rotation of the body.
- If the body is immovable, then you can ignore any change in position and rotation.

Parsing 3D Model Files

All objects in TekPhysics must be read from a 3D model file. These are specified by a simple format that allows them to be easily written by hand or converted from other formats if needed, while also closely resembling the data that is needed for rendering them with OpenGL. The general rules go like this:

- All lines beginning with a hash (#) are ignored and treated as comments

- The file is divided into three sections which can take any order. They are the vertices, indices and layout sections. They are specified by writing a line that contains only the capitalised version of the section name.
- Any numbers following the section name are added to a list under that section.
- The vertices section contains rational numbers, while the indices and layout sections must contain integers.
- The vertices section contains the per-vertex data of the model.
- The indices section contains the order of drawing vertices to create triangles.
- The layout section contains the way of “chunking” vertices, for example a layout of “3 3” tells us that each vertex is 6 numbers long, split into two three-component vectors.
- The wildcard “\$POSITION_LAYOUT_INDEX n” describes where the position data for each vertex is found. It allows the model file to be used by both graphics and physics. The layout index should match the layout index used in the shader program.

In order to read these files, you will need to implement a program that can understand these rules. For example, this program can read the model files:

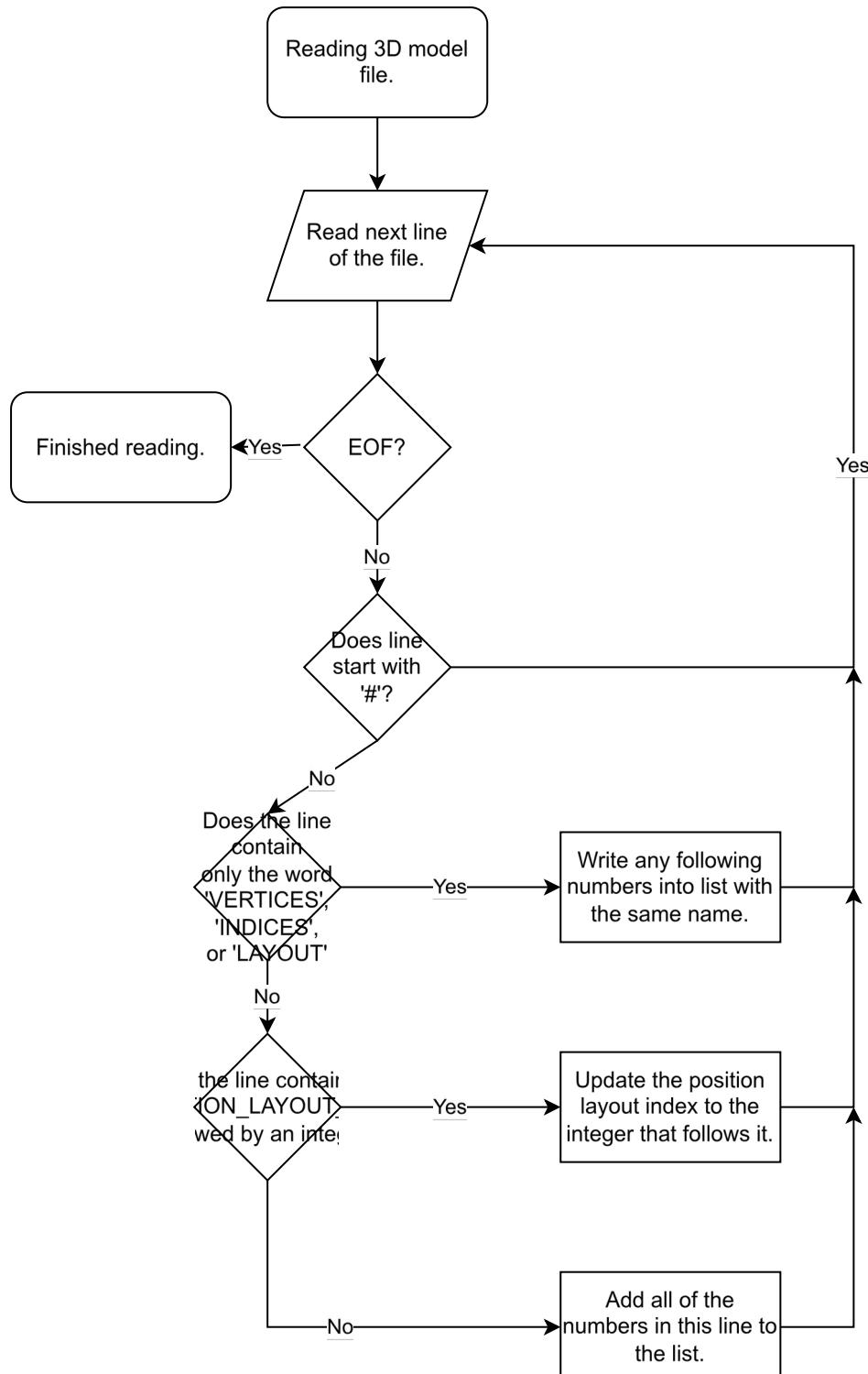


Figure 26: A flowchart describing a way to read a model file.

An example of a simple model file for a cube would be something like this:

```
1 # format is (x y z) (normal_x normal_y normal_z)
2 VERTICES
3 -1.0  1.0  -1.0  0.0  1.0  0.0
```

```

4 -1.0 1.0 1.0 0.0 1.0 0.0
5 1.0 1.0 1.0 0.0 1.0 0.0
6 1.0 1.0 -1.0 0.0 1.0 0.0
7
8 1.0 -1.0 -1.0 1.0 0.0 0.0
9 1.0 1.0 -1.0 1.0 0.0 0.0
10 1.0 1.0 1.0 1.0 0.0 0.0
11 1.0 -1.0 1.0 1.0 0.0 0.0
12
13 1.0 -1.0 1.0 0.0 0.0 1.0
14 1.0 1.0 1.0 0.0 0.0 1.0
15 -1.0 1.0 1.0 0.0 0.0 1.0
16 -1.0 -1.0 1.0 0.0 0.0 1.0
17
18 -1.0 -1.0 -1.0 -1.0 0.0 0.0
19 -1.0 -1.0 1.0 -1.0 0.0 0.0
20 -1.0 1.0 1.0 -1.0 0.0 0.0
21 -1.0 1.0 -1.0 -1.0 0.0 0.0
22
23 -1.0 -1.0 -1.0 0.0 0.0 -1.0
24 -1.0 1.0 -1.0 0.0 0.0 -1.0
25 1.0 1.0 -1.0 0.0 0.0 -1.0
26 1.0 -1.0 -1.0 0.0 0.0 -1.0
27
28 -1.0 -1.0 1.0 0.0 -1.0 0.0
29 -1.0 -1.0 -1.0 0.0 -1.0 0.0
30 1.0 -1.0 -1.0 0.0 -1.0 0.0
31 1.0 -1.0 1.0 0.0 -1.0 0.0
32
33 INDICES
34 0 1 2
35 0 2 3
36 4 5 6
37 4 6 7
38 8 9 10
39 8 10 11
40 12 13 14
41 12 14 15
42 16 17 18
43 16 18 19
44 20 21 22
45 20 22 23
46

```

```

47 LAYOUT
48 3 3
49
50 $POSITION_LAYOUT_INDEX 0

```

Velocity and Angular Velocity Integration

In each physics step, the position of each body changes due to its velocity, and its rotation changes due to its angular velocity. Calculating this change is quite simple because it's a simple integration, velocity is the rate of change of displacement, and angular velocity is the rate of change of angle. Therefore, we can integrate the velocity and angular velocity functions between the previous time and the current time in the simulation to retrieve the change in position and angle respectively. However, because the physics time is so small, we can approximate this integration using the midpoint rule. In other words, at each time step:

- Position = Position + Velocity * Time Difference
- Rotation = Rotation + Angular Velocity * Time Difference

A complexity arises when calculating the final angle. This is because angular velocity is a 3 component vector which represents an axis of rotation, where the magnitude of this vector is the rate of change of angle around this axis in radians per second. However, the actual rotation is a quaternion / 4 component vector. To solve this, we simply find the magnitude of the angular velocity, multiply that by the time difference, and then rotate the quaternion about the axis by the calculated amount. If the change in angle is insignificant (e.g. less than 1e-5) then the rotation should not occur, as this could induce floating point instability into the simulation, as the numbers may be too small to be accurately represented.

An optimisation can be made by ignoring any bodies that are immovable from this step, in fact they should definitely be ignored to avoid accidental moving of an immovable object.

Data Structures

Within the program, a variety of data structures will be required in different parts of the code in order to allow the algorithms to function correctly. These data structures will need to be implemented first to allow them to be used across the program. The data structures that need to be implemented are listed below.

Data Structure	Description	Usage
Bit Set	An array-like structure that is used to store a set of boolean values. It is more memory efficient than an array as it uses each bit in a block	Used primarily within the expanding polytope algorithm to track the presence of edges in the polytope.

	of memory to store a true/false value rather than using an array of integers that could use 32 or 64 bits to store a single true or false value.	
Hash Table	An unordered set of key-value pairs. The key is hashed to get an array index, and the data is stored at that index in the array. Then, to retrieve the data, the key is hashed again to find the array index to get the value.	Used extensively throughout the program. Some notable uses include the YAML file parser which encodes the file into hash tables. Also used to cache model and material files so that they are not needlessly loaded multiple times for each object that uses that particular file.
List (Linked List)	A set of nodes that contain some data and a pointer to the next node in the list. This provides some benefits as opposed to a typical array as it allows for much easier manipulation of items in the list, especially for inserting and deleting items.	Lists are also used widely in the program, mainly to store callbacks for the program. When the window receives a callback such as a mouse click, only one function can be designated in GLFW to receive this callback. To allow for multiple callbacks to receive the event, a list is created to store multiple function pointers, then the list is iterated over to push this event to each function in the list of callbacks. There are many more uses, this is the first that came to mind.
Priority Queue	A queue data structure that takes in data as well as an associated priority, this priority will determine which data is dequeued first.	Used when loading window layouts from a file. Windows are stored as yml files which are loaded as hash tables. Hash tables are unordered, this means there is no way to maintain the order of the items in the window. An "index" is therefore added to each item in the window. For robustness, the index is only used to sort the items, not index them directly. The priority queue is used to sort the items into the correct order. This also allows new items to be inserted without having to scroll through the file to place it in the order.
Queue	A typical queue structure, the first items added will be the first items to be dequeued.	Used in the scenario code to store unused object IDs. When an object is deleted, it leaves a gap in the object

		IDs which are assigned sequentially. When an object is deleted, its ID is added to the list of unused IDs, which is then checked each time a new object is created. This allows the IDs to be reused.
Stack	A typical stack data structure, the first items to be added will be the last items to be popped.	Used in the parsing of YAML files. YAML relies heavily on indentation to describe the relationship between data. In order to check that indentation is correct, the amount of indentation is stored in a stack. When the indentation is reduced, the stack is popped until the current indent matches the last popped indent, this tells us if this indentation change is valid, and how many parents we have gone up by.
Thread Queue	A single-producer single-consumer queue that works across two threads and maintains data integrity.	The thread queue is used to establish communication between the physics engine thread and the window/graphics thread.
Vector	An automatically resizing array that allocates more memory as needed. Also provides some useful functionality such as removing items and “shuffling” the other items down the list to fill the gap, popping items from the end, inserting items in the middle.	The vector is possibly the most used data structure in the program. It is used to store the objects in the scene. It is also used as a stack when iterating through the collision structure to check for collisions, as it is much quicker than the other stack structure which uses linked-list style nodes.

Much more detailed explanation can be found below in the key algorithm design, where you can see the exact function calls that will allow the data structures to work within the algorithms, which ones are used in which algorithm, and how they work with different programming concepts such as iteration and selection.

TekPhys Key Algorithms

Collider Structure Generation

The “collider structure” is a binary tree containing different types of colliders. Each node of the tree is an oriented bounding box (OBB), with the root node being an OBB that encompasses the entire rigid body the structure is being generated for. Each successive child is then an OBB encompassing half of the

aforementioned body's polygons, continually dividing until the body cannot be split further – this happens when the individual polygons are reached. When this happens, a leaf node is added containing the polygon itself.

In order to determine the shape of the OBB that best surrounds a set of points, we can analyse the distribution using a covariance matrix. This essentially describes how the points are spread out in space. We can find the eigenvectors of this matrix to determine the three orthogonal vectors of maximum spread – these become the three main axes of the OBB.

To find the eigenvectors of a matrix, the LAPACK package can be used. This is a collection of linear algebra subroutines written in Fortran that have been highly optimised to run efficiently. The subroutine required for finding eigenvectors (and eigenvalues) is "ssyev()", where:

- "s" – Single precision floating point number.
- "sy" – Symmetric matrix.
- "ev" – Find eigenvectors / eigenvalues.

Below is a pseudocode implementation of this algorithm:

```

1  FUNCTION CreateCollider(body)
2      stack ← NEW STACK
3      triangles ← NEW ARRAY
4
5      // generate triangles from mesh data
6      triangles ← GenerateTriangleArray(body.vertices,
body.indices)
7
8      // create root collider node containing all triangles
9      rootNode ← CreateColliderNode(0, triangles,
ALL_INDICES(triangles))
10
11     Push(stack, rootNode)
12
13     // build binary tree
14     WHILE NOT IsEmpty(stack) DO
15         currentNode ← Pop(stack)
16
17         // partition triangles based on OBB axis
18         axis ← FindDividingAxis(currentNode, triangles)
19
20         IF axis = NONE THEN
21             // cannot split further -> make a leaf
22             ConvertToLeaf(currentNode, triangles)

```

```

23         ELSE
24             // split into left and right sets of indices
25             (leftIndices, rightIndices) ←
PartitionTriangles(currentNode, axis, triangles)
26
27             // create left child
28             leftNode ← CreateColliderNode(triangles,
leftIndices)
29             currentNode.left ← leftNode
30             Push(stack, leftNode)
31
32             // create right child
33             rightNode ← CreateColliderNode(triangles,
rightIndices)
34             currentNode.right ← rightNode
35             Push(stack, rightNode)
36         ENDIF
37     ENDWHILE
38
39     RETURN rootNode
40 ENDFUNCTION
41
42 FUNCTION CreateColliderNode(type, id, triangles, indices)
43     node ← NEW ColliderNode
44     node.id ← id
45     node.type ← type
46     node.indices ← indices
47
48     // build bounding box
49     node.obb ← CreateOBB(triangles, indices)
50
51     RETURN node
52 ENDFUNCTION
53
54
55 FUNCTION CreateOBB(triangles, indices)
56     mean ← CalculateConvexHullMean(triangles, indices)
57     covariance ← CalculateCovarianceMatrix(triangles, indices,
mean)
58
59     // eigenvectors give main axes of spread
60     (eigenvectors, eigenvalues) ←
CalculateEigenvectors(covariance)

```

```

61
62      // project triangles onto axes to find extents
63      FOR i ← 0 TO 2
64          (minProj, maxProj) ← FindProjections(triangles, indices,
eigenvectors[i])
65          obb.axes[i] ← eigenvectors[i]
66          obb.halfExtent[i] ← (maxProj - minProj) / 2
67      ENDFOR
68
69      obb.centre ← CalculateCentre(eigenvectors, minProj, maxProj)
70
71      RETURN obb
72  ENDFUNCTION

```

Rigid Body Properties Generation

The program will be required to calculate the properties of a rigid body including the mass, centre of mass and the inertia tensor. It does this by selecting an arbitrary point in space (the algorithm uses the origin), and using this point along with each triangle that makes up the body to form a set of tetrahedra. As a tetrahedra is a simplex (simplest shape in n dimensional space), we can take its centre of mass to be the average of its vertices. By calculating the volume using known formulae, we can then multiply by the density of the body to find the mass, and additionally find the inertia tensor for each tetrahedron, and then combine these values to find the properties of the whole body.

```

1  FUNCTION CalculateBodyProperties(body)
2      DECLARE tetrahedronData ← ARRAY OF TetrahedronData
3      DECLARE origin ← (0, 0, 0)
4
5      DECLARE totalVolume ← 0
6      DECLARE weightedSum ← (0, 0, 0)
7
8      // first pass, compute volume and centre of mass
9      FOR i ← 0 TO body.numIndices - 1 STEP 3
10         // vertices of triangle
11         a ← body.vertices[ body.indices[i] ]
12         b ← body.vertices[ body.indices[i + 1] ]
13         c ← body.vertices[ body.indices[i + 2] ]
14
15         // form tetrahedron with origin
16         tetraVolume ← SignedTetrahedronVolume(origin, a, b, c)
17
18         totalVolume ← totalVolume + tetraVolume
19

```

```

20      // centroid of tetrahedron is average of its 4 vertices
21      tetraCentroid ← (origin + a + b + c) / 4
22
23      // weighted sum for global centre of mass
24      weightedSum ← weightedSum + (tetraCentroid ×
25      tetraVolume)
26
27      // store data for later use
28      APPEND tetrahedronData, (tetraVolume, tetraCentroid)
29      ENDFOR
30
31      // overall centre of mass = weighted average
32      body.centreOfMass ← weightedSum / totalVolume
33
34      // mass = density × absolute volume
35      body.volume ← ABS(totalVolume)
36      body.mass ← body.volume × body.density
37
38      // second pass, compute inertia tensor
39      DECLARE inertiaTensor AS 3×3 zero matrix
40
41      FOR i ← 0 TO body.numIndices - 1 STEP 3
42          a ← body.vertices[ body.indices[i] ]
43          b ← body.vertices[ body.indices[i + 1] ]
44          c ← body.vertices[ body.indices[i + 2] ]
45
46          index ← i / 3
47          tetraMass ← ABS(tetrahedronData[index].volume ×
48          body.density)
49
50          // inertia tensor of this tetrahedron about its own
51          // centroid
52          tetraInertia ← TetrahedronInertiaTensor(origin, a, b, c,
53          tetraMass)
54
55          // translation vector from object CoM to tetrahedron CoM
56          translate ← tetrahedronData[index].centroid -
body.centreOfMass
57
58          // adjust inertia tensor using parallel axis theorem
59          tetraInertia ← TranslateInertiaTensor(tetraInertia,
60          tetraMass, translate)
61
62

```

```

57      // accumulate result
58      inertiaTensor ← inertiaTensor + tetraInertia
59  ENDFOR
60
61      // store inverse inertia tensor
62      body.inverseInertiaTensor ← InverseMatrix(inertiaTensor)
63  ENDFUNCTION

```

Triangle-Triangle Collision Detection (GJK)

In order to detect a collision between two triangles, the GJK (Gilbert, Johnson and Keerthi) algorithm should be implemented. The algorithm is based on the principle of the Minkowski Sum (also known as the Minkowski Difference), which is the idea of “adding and subtracting shapes” by finding the difference between every combination of points on each shape to create a new shape. If the Minkowski difference contains the point (0, 0), it means that at least somewhere, the shapes have a common point. It would be unwise to actually try every single point, so instead we iteratively build a simplex of points on the Minkowski difference that aims to quickly find a simplex that contains the origin. Once a simplex is found that contains the origin, there must be a collision. If it becomes impossible to create such a simplex, there is no collision. Pseudocode for this algorithm can be found below:

```

1  FUNCTION TriangleFurthestPoint(triangle, direction)
2      maxIndex ← 0
3      maxValue ← -∞
4
5      FOR i ← 0 TO 2
6          value ← DOT(triangle[i], direction)
7          IF value > maxValue THEN
8              maxIndex ← i
9              maxValue ← value
10         ENDIF
11     NEXT
12
13     RETURN maxIndex
14 ENDFUNCTION
15
16 FUNCTION TriangleSupport(triangleA, triangleB, direction)
17     opposite ← -direction
18     indexA ← TriangleFurthestPoint(triangleA, direction)
19     indexB ← TriangleFurthestPoint(triangleB, opposite)
20
21     point.support ← triangleA[indexA] - triangleB[indexB]
22     point.a ← triangleA[indexA]

```

```

23     point.b ← triangleB[indexB]
24
25     RETURN point
26 ENDFUNCTION
27
28 PROCEDURE UpdateLineSimplex(OUT direction, simplex)
29     ao ← -simplex[0].support
30     ab ← simplex[1].support - simplex[0].support
31
32     direction ← CROSS(ab, ao)
33     direction ← CROSS(direction, ab)
34 ENDPROCEDURE
35
36 PROCEDURE UpdateTriangleSimplexLineAB(ao, ab, OUT direction,
OUT lenSimplex)
37     IF DOT(ab, ao) > 0 THEN
38         direction ← CROSS(ab, ao)
39         direction ← CROSS(direction, ab)
40         lenSimplex ← 2
41     ELSE
42         direction ← ao
43         lenSimplex ← 1
44     ENDIF
45 ENDPROCEDURE
46
47 PROCEDURE UpdateTriangleSimplex(OUT direction, simplex, INOUT
lenSimplex)
48     ao ← -simplex[0].support
49     ab ← simplex[1].support - simplex[0].support
50     ac ← simplex[2].support - simplex[0].support
51
52     normal ← CROSS(ab, ac)
53     perpAC ← CROSS(normal, ac)
54
55     IF DOT(perpAC, ao) > 0 THEN
56         IF DOT(ac, ao) > 0 THEN
57             direction ← CROSS(ac, ao)
58             direction ← CROSS(direction, ac)
59             simplex[1] ← simplex[2]
60             lenSimplex ← 2
61         ELSE
62             UpdateTriangleSimplexLineAB(ao, ab, direction,
lenSimplex)

```

```

63         ENDIF
64     ELSE
65         perpAB ← CROSS(ab, normal)
66         IF DOT(perpAB, ao) > 0 THEN
67             UpdateTriangleSimplexLineAB(ao, ab, direction,
lenSimplex)
68         ELSE
69             IF DOT(normal, ao) > 0 THEN
70                 direction ← normal
71             ELSE
72                 direction ← -normal
73                 SWAP(simplex[1], simplex[2])
74             ENDIF
75         ENDIF
76     ENDIF
77 ENDPROCEDURE
78
79 FUNCTION UpdateTetrahedronSimplex(OUT direction, simplex, INOUT
lenSimplex)
80     ao ← -simplex[0].support
81     ab ← simplex[1].support - simplex[0].support
82     ac ← simplex[2].support - simplex[0].support
83     ad ← simplex[3].support - simplex[0].support
84
85     normABC ← CROSS(ab, ac)
86     IF DOT(normABC, ao) > 0 THEN
87         lenSimplex ← 3
88         UpdateTriangleSimplex(direction, simplex, lenSimplex)
89         RETURN FALSE
90     ENDIF
91
92     normACD ← CROSS(ac, ad)
93     IF DOT(normACD, ao) > 0 THEN
94         simplex[1] ← simplex[2]
95         simplex[2] ← simplex[3]
96         lenSimplex ← 3
97         UpdateTriangleSimplex(direction, simplex, lenSimplex)
98         RETURN FALSE
99     ENDIF
100
101    normADB ← CROSS(ad, ab)
102    IF DOT(normADB, ao) > 0 THEN
103        simplex[2] ← simplex[1]

```

```

104         simplex[1] ← simplex[3]
105         lenSimplex ← 3
106         UpdateTriangleSimplex(direction, simplex, lenSimplex)
107         RETURN FALSE
108     ENDIF
109
110     // origin is inside the tetrahedron
111     RETURN TRUE
112 ENDFUNCTION
113
114 FUNCTION UpdateSimplex(OUT direction, simplex, INOUT
lenSimplex)
115     SWITCH lenSimplex
116         CASE 2:
117             UpdateLineSimplex(direction, simplex)
118         CASE 3:
119             UpdateTriangleSimplex(direction, simplex,
lenSimplex)
120         CASE 4:
121             RETURN UpdateTetrahedronSimplex(direction, simplex,
lenSimplex)
122     ENDSWITCH
123     RETURN FALSE
124 ENDFUNCTION
125 FUNCTION CheckTriangleCollision(triangleA, triangleB)
126     // initial direction between centroids
127     centroidA ← (triangleA[0] + triangleA[1] + triangleA[2]) /
3
128     centroidB ← (triangleB[0] + triangleB[1] + triangleB[2]) /
3
129     direction ← centroidB - centroidA
130
131     normalA ← TriangleNormal(triangleA)
132     normalB ← TriangleNormal(triangleB)
133
134     // check if triangles are coplanar or have the same centre
135     IF |direction| ≈ 0 OR
136         |DOT(direction, normalA)| ≈ 0 OR
137         |DOT(direction, normalB)| ≈ 0 THEN
138         direction ← RANDOM_UNIT_VECTOR()
139     ENDIF
140
141     // first simplex point

```

```

142      simplex[0] ← TriangleSupport(triangleA, triangleB,
direction)
143      lenSimplex ← 1
144
145      IF DOT(simplex[0].support, direction) < 0 THEN
146          RETURN FALSE
147      ENDIF
148
149      direction ← -direction
150      support ← TriangleSupport(triangleA, triangleB, direction)
151      separation ← 0
152
153      // main iteration loop
154      WHILE DOT(support.support, direction) ≥ 0
155          separation ← DOT(support.support, direction)
156
157          SHIFT simplex right by 1
158          simplex[0] ← support
159          lenSimplex ← lenSimplex + 1
160
161          IF UpdateSimplex(direction, simplex, lenSimplex) = TRUE
THEN
162              RETURN TRUE
163          ENDIF
164
165          IF |direction| < EPSILON THEN
166              RETURN TRUE
167          ENDIF
168
169          support ← TriangleSupport(triangleA, triangleB,
direction)
170          NORMALISE(direction)
171      ENDWHILE
172
173      RETURN FALSE
174  ENDFUNCTION

```

Triangle-Triangle Collision Contact Points (EPA)

Once a collision has been detected between triangles, we need to find the points of contact between them. To do this we use the Expanding Polytope Algorithm (EPA). This algorithm takes over from the simplex created by GJK and ensures that it is not degenerate (not a line or flat plane). Then, it recursively searches for the closest face to the origin, finds the closest point on the Minkowski

difference in the direction of the face's normal, and removes the face. Then, using the point found, it adds new faces to "patch the hole" by removing the face. This is repeated until the newly found point cannot get any closer to the origin. The distance from the origin to this point is the contact depth, and the direction to this point is the contact normal.

```

1  FUNCTION GetMinAxis(vector) RETURNS INTEGER
2      min_axis ← 0
3      min_length ← +INFINITY
4      FOR i ← 0 TO 2
5          IF ABS(vector[i]) < min_length THEN
6              min_length ← ABS(vector[i])
7              min_axis ← i
8          ENDIF
9      NEXT i
10     RETURN min_axis
11 ENDFUNCTION
12
13 FUNCTION GetClosestFace(vertices, faces, face_index,
face_distance, face_normal) RETURNS BOOLEAN
14     face_distance ← +INFINITY
15     face_index ← 0
16
17     FOR i ← 0 TO LENGTH(faces) - 1
18         indices ← faces[i]           # indices = [a,b,c]
19         vertex_a ← vertices[indices[0]]
20         vertex_b ← vertices[indices[1]]
21         vertex_c ← vertices[indices[2]]
22
23         ab ← vertex_b.support - vertex_a.support
24         ac ← vertex_c.support - vertex_a.support
25         normal ← CROSS(ab, ac)
26
27         mag_normal ← NORM(normal)
28         IF mag_normal < EPSILON THEN
29             CONTINUE FOR
30         ENDIF
31
32         normal ← normal / mag_normal
33         curr_distance ← ABS(DOT(normal, vertex_a.support))
34

```

```

35      IF curr_distance < face_distance THEN
36          face_distance ← curr_distance
37          face_index ← i
38          face_normal ← normal
39      ENDIF
40      NEXT i
41
42      RETURN TRUE
43 ENDFUNCTION
44
45 PROCEDURE RemoveAllVisibleFaces(vertices, faces, edges,
edges_bitset, support)
46     i ← 0
47     WHILE i < LENGTH(faces)
48         indices ← faces[i]
49
50         vertex_a ← vertices[indices[0]]
51         vertex_b ← vertices[indices[1]]
52         vertex_c ← vertices[indices[2]]
53
54         ab ← vertex_b.support - vertex_a.support
55         ac ← vertex_c.support - vertex_a.support
56         normal ← CROSS(ab, ac)
57         mag_normal ← NORM(normal)
58
59         IF mag_normal < EPSILON THEN
60             REMOVE faces[i]
61             i ← i - 1
62             CONTINUE WHILE
63         ENDIF
64
65         normal ← normal / mag_normal
66         proj ← support - vertex_a.support
67         orientation ← DOT(normal, proj)
68
69         IF orientation > EPSILON THEN
70             REMOVE faces[i]
71             i ← i - 1
72             FOR j ← 0 TO 2
73                 edge_a ← indices[j]

```

```

74             edge_b ← indices[(j + 1) MOD 3]
75             RemoveEdgeFromPolytope(edges, edges_bitset,
[edge_a, edge_b])
76             NEXT j
77         ENDIF
78
79         i ← i + 1
80     ENDWHILE
81 ENDPROCEDURE
82
83 PROCEDURE AddAllFillerFaces(vertices, faces, edges,
edges_bitset, support_index)
84     FOR i ← 0 TO LENGTH(edges) - 1
85         indices ← edges[i]
86         has_edge ← edges_bitset[indices[0]][indices[1]]
87         IF has_edge THEN
88             AddFillerFace(vertices, faces, indices,
support_index)
89         ENDIF
90     NEXT i
91 ENDPROCEDURE
92
93 PROCEDURE ProjectOriginToBarycentric(a, b, c, barycentric)
94     ab ← b - a
95     ac ← c - a
96     normal ← CROSS(ab, ac)
97     ao ← -a
98
99     ab_ao ← CROSS(ab, ao)
100    ao_ac ← CROSS(ao, ac)
101    square_normal ← DOT(normal, normal)
102
103    barycentric[2] ← DOT(ab_ao, normal) / square_normal
104    barycentric[1] ← DOT(ao_ac, normal) / square_normal
105    barycentric[0] ← 1 - barycentric[2] - barycentric[1]
106 ENDPROCEDURE
107
108 PROCEDURE CreatePointFromBarycentric(a, b, c, barycentric,
point)
109    point ← a * barycentric[0]

```

```

110      point ← point + b * barycentric[1]
111      point ← point + c * barycentric[2]
112  ENDPROCEDURE
113
114  FUNCTION GetTriangleCollisionPoints(triangle_a, triangle_b,
simplex,
115                                         contact_depth,
contact_normal,
116                                         contact_a, contact_b)
RETURNS BOOLEAN
117
118      # initialise buffers
119      vertices ← []
120      faces ← []
121      edges ← []
122      edge_bitset ← 2DARRAY(FALSE)
123
124      FOR i ← 0 TO 3
125          APPEND vertices, simplex[i]
126      NEXT i
127
128      # add initial tetrahedron faces
129      APPEND faces, [0,1,2]
130      APPEND faces, [0,3,1]
131      APPEND faces, [0,2,3]
132      APPEND faces, [1,3,2]
133
134      face_index ← 0
135      face_normal ← (0,0,0)
136      min_distance ← +INFINITY
137      num_iterations ← 0
138
139      WHILE min_distance = +INFINITY
140          edges ← []
141          CLEAR(edge_bitset)
142
143          GetClosestFace(vertices, faces, face_index,
min_distance, face_normal)
144
145          IF num_iterations > 20 THEN

```

```

146          BREAK WHILE
147      ENDIF
148
149      support ← TriangleSupport(triangle_a, triangle_b,
face_normal)
150
151      IF ABS(DOT(face_normal, support.support) -
min_distance) < EPSILON THEN
152          CONTINUE WHILE
153      ENDIF
154
155      min_distance ← +INFINITY
156      RemoveFaceFromPolytope(faces, edges, edge_bitset,
face_index)
157      RemoveAllVisibleFaces(vertices, faces, edges,
edge_bitset, support.support)
158
159      support_index ← LENGTH(vertices)
160      APPEND vertices, support
161      AddAllFillerFaces(vertices, faces, edges, edge_bitset,
support_index)
162
163      num_iterations ← num_iterations + 1
164  ENDWHILE
165
166  # compute final contact info
167  indices ← faces[face_index]
168  vertex_a ← vertices[indices[0]]
169  vertex_b ← vertices[indices[1]]
170  vertex_c ← vertices[indices[2]]
171
172  DECLARE barycentric[3]
173  ProjectOriginToBarycentric(vertex_a.support,
vertex_b.support, vertex_c.support, barycentric)
174
175  CreatePointFromBarycentric(vertex_a.a, vertex_b.a,
vertex_c.a, barycentric, contact_a)
176  CreatePointFromBarycentric(vertex_a.b, vertex_b.b,
vertex_c.b, barycentric, contact_b)
177

```

```

178     contact_normal ← face_normal
179     contact_depth ← min_distance
180
181     RETURN TRUE
182 ENDFUNCTION

```

Triangle-Oriented Bounding Box (OBB) Collision Detection

This algorithm works by using a matrix transformation to align the OBB with the coordinate axes (turning into an AABB), and then applying the same transform to the triangle in order to maintain the distances between the colliders. This allows a simplified Separating Axis Theorem (SAT) test to happen between the AABB and the triangle – project the triangle and the AABB one at a time onto a set of axes, and check if they overlap at any point. If an overlap is found, then there is a collision.

```

1  FUNCTION CreateOBBTransform(obb)
2      // construct transform matrix from obb orientation and
centre
3      tempMatrix ← [
4          obb.w_axes[0][0], obb.w_axes[0][1], obb.w_axes[0][2], 0,
5          obb.w_axes[1][0], obb.w_axes[1][1], obb.w_axes[1][2], 0,
6          obb.w_axes[2][0], obb.w_axes[2][1], obb.w_axes[2][2], 0,
7          obb.w_centre[0], obb.w_centre[1], obb.w_centre[2], 1
8      ]
9      RETURN InverseMatrix(tempMatrix)
10 ENDFUNCTION
11
12 FUNCTION CheckAABBTriangleCollision(halfExtents[3], triangle[3])
13     // build triangle edge vectors
14     FOR i ← 0 TO 2
15         faceVectors[i] ← triangle[(i + 1) MOD 3] - triangle[i]
16     ENDFOR
17
18     DECLARE xyzAxes ← { (1,0,0), (0,1,0), (0,0,1) }
19
20     // test axis = triangle normal
21     faceNormal ← CrossProduct(faceVectors[0], faceVectors[1])
22     Normalize(faceNormal)
23
24     test ← ABS(DotProduct(triangle[0], faceNormal))
25     projection ← 0
26     FOR i ← 0 TO 2

```

```

27         projection ← projection + halfExtents[i] ×
ABS(faceNormal[i])
28     ENDFOR
29     IF test > projection THEN
30         RETURN FALSE
31     ENDIF
32
33     // test axes = X, Y, Z (aabb axes)
34     FOR i ← 0 TO 2
35         FOR j ← 0 TO 2
36             dots[j] ← DotProduct(triangle[j], xyzAxes[i])
37         ENDFOR
38
39         triMin ← Min(dots[0], dots[1], dots[2])
40         triMax ← Max(dots[0], dots[1], dots[2])
41
42         IF triMin > halfExtents[i] OR triMax < -halfExtents[i]
THEN
43             RETURN FALSE
44         ENDIF
45     ENDFOR
46
47     // test axes = cross products of AABB axes and triangle
edges
48     FOR i ← 0 TO 2
49         FOR j ← 0 TO 2
50             currAxis ← CrossProduct(xyzAxes[i], faceVectors[j])
51
52             projection ← 0
53             FOR k ← 0 TO 2
54                 projection ← projection + halfExtents[k] ×
ABS(DotProduct(xyzAxes[k], currAxis))
55                 dots[k] ← DotProduct(triangle[k], currAxis)
56             ENDFOR
57
58             triMin ← Min(dots[0], dots[1], dots[2])
59             triMax ← Max(dots[0], dots[1], dots[2])
60
61             IF triMin > projection OR triMax < -projection THEN
62                 RETURN FALSE

```

```

63         ENDIF
64     ENDFOR
65 ENDFOR
66
67 // if no separating axis found, collision exists
68 RETURN TRUE
69 ENDFUNCTION
70
71 FUNCTION CheckOBBTriangleCollision(obb, triangle[3])
72     transform ← CreateOBBTransform(obb)
73
74     FOR i ← 0 TO 2
75         transformedTriangle[i] ← MultiplyMatrixVector(transform,
76 triangle[i])
77     ENDFOR
78
79     RETURN CheckAABBTriangleCollision(obb.w_half_extents,
80 transformedTriangle)
81 ENDFUNCTION
82
83 FUNCTION CheckOBBTrianglesCollision(obb, triangles[], numTriangles)
84     FOR i ← 0 TO numTriangles - 1
85         tri ← { triangles[i*3], triangles[i*3 + 1],
86 triangles[i*3 + 2] }
87         IF CheckOBBTriangleCollision(obb, tri) = TRUE THEN
88             RETURN TRUE
89         ENDIF
90     ENDFOR
91     RETURN FALSE
92 ENDFUNCTION

```

OBB-OBB Collision Detection

This algorithm is based on the one written in Gottstall's "OBB-Tree" paper. It is essentially just an optimised SAT test specific to OBBs. It checks against 15 axes, 6 to account for the normals to each OBB, and then 9 axes which are the cross products of the face normals.

```

1 FUNCTION CheckOBBCollision(obbA, oobbB)
2     // vector from centre of A to centre of B
3     translate ← oobbB.centre - oobbA.centre

```

```

4
5      // precompute dot products
6      FOR i ← 0 TO 2
7          tArray[i] ← DotProduct(translate, obbA.axes[i])
8          FOR j ← 0 TO 2
9              dotMatrix[i][j] ← DotProduct(obbA.axes[i],
10                 obbB.axes[j]) + EPSILON
11             ENDFOR
12         ENDFOR
13
14         // test OBB As face normals
15         FOR i ← 0 TO 2
16             magSum ← 0
17             FOR j ← 0 TO 2
18                 magSum ← magSum + ABS(obbB.halfExtents[j] ×
dotMatrix[i][j])
19             ENDFOR
20             IF ABS(tArray[i]) > obbA.halfExtents[i] + magSum THEN
21                 RETURN FALSE
22             ENDIF
23         ENDFOR
24
25         // test OBB Bs face normals
26         FOR i ← 0 TO 2
27             magSum ← 0
28             FOR j ← 0 TO 2
29                 magSum ← magSum + ABS(obbA.halfExtents[j] ×
dotMatrix[j][i])
30             ENDFOR
31             projection ← ABS(DotProduct(translate, obbB.axes[i]))
32             IF projection > obbB.halfExtents[i] + magSum THEN
33                 RETURN FALSE
34             ENDIF
35         ENDFOR
36
37         // test cross products of axes (edge vs edge)
38         multisIndices ← { {2,1}, {0,2}, {1,0} }
39         cycIndices ← { {1,2}, {0,2}, {0,1} }
40

```

```

41      FOR i ← 0 TO 2
42          FOR j ← 0 TO 2
43              tiA ← multisIndices[i][0]
44              tiB ← multisIndices[i][1]
45
46              cmpBase ← tArray[tiA] × DotProduct(obbA.axes[tiB],
obbB.axes[j])
47              cmpSubt ← tArray[tiB] × DotProduct(obbA.axes[tiA],
obbB.axes[j])
48              cmp ← ABS(cmpBase - cmpSubt)
49
50              cycLL ← cycIndices[i][0]
51              cycLH ← cycIndices[i][1]
52              cycSL ← cycIndices[j][0]
53              cycSH ← cycIndices[j][1]
54
55              tst ← 0
56              tst ← tst + ABS(obbA.halfExtents[cycLL] ×
dotMatrix[cycLH][j])
57              tst ← tst + ABS(obbA.halfExtents[cycLH] ×
dotMatrix[cycLL][j])
58              tst ← tst + ABS(obbB.halfExtents[cycSL] ×
dotMatrix[i][cycSH])
59              tst ← tst + ABS(obbB.halfExtents[cycSH] ×
dotMatrix[i][cycSL])
60
61              IF cmp > tst THEN
62                  RETURN FALSE
63              ENDIF
64          ENDFOR
65      ENDFOR
66
67      // if no separating axis is found, collision must exist
68      RETURN TRUE
69  ENDFUNCTION

```

Collider-Collider Collision Detection

This algorithm is responsible for checking if two collision structures are colliding. It works by initialising a stack and adding the root nodes of each collision structure as a pair. Then the algorithm repeatedly checks for a collision between the children of these nodes if they are not leaf nodes, adding each colliding pair

back to the stack. If a collision is found between two leaf nodes, then a collision manifold is created which is added to a list of collision manifolds for this collision. If the stack becomes empty before a collision is found, then there must be no collision. Otherwise, the list of collision manifolds is returned.

```

1  FUNCTION GetCollisionManifolds(bodyA, bodyB)
2      DECLARE manifolds ← ARRAY OF Manifold
3      collision ← FALSE
4
5      // initialise stack with root collider nodes
6      stack ← EmptyStack()
7      Push(stack, (bodyA.collider, bodyB.collider))
8
9      WHILE NOT IsEmpty(stack)
10         (nodeA, nodeB) ← Pop(stack)
11
12         // update bounding boxes for non-leaf nodes
13         IF nodeA.type = COLLIDER_NODE THEN
14             UpdateOBB(nodeA.left.obb, bodyA.transform)
15             UpdateOBB(nodeA.right.obb, bodyA.transform)
16         ENDIF
17         IF nodeB.type = COLLIDER_NODE THEN
18             UpdateOBB(nodeB.left.obb, bodyB.transform)
19             UpdateOBB(nodeB.right.obb, bodyB.transform)
20         ENDIF
21
22         // try all child combinations (2x2 = 4 possibilities)
23         FOR i ← 0 TO 1
24             FOR j ← 0 TO 1
25                 IF nodeA.type = COLLIDER_NODE THEN
26                     childA ← GetChild(nodeA, i)
27                 ELSE
28                     childA ← nodeA
29                 ENDIF
30
31                 IF nodeB.type = COLLIDER_NODE THEN
32                     childB ← GetChild(nodeB, j)
33                 ELSE
34                     childB ← nodeB
35                 ENDIF
36

```

```

37             subCollision ← FALSE
38
39             // both are OBB nodes
40             IF childA.type = COLLIDER_NODE AND childB.type =
COLLIDER_NODE THEN
41                 subCollision ← CheckOBBCollision(childA.obb,
childB.obb)
42
43                 // OBB vs Leaf
44                 ELSE IF childA.type = COLLIDER_NODE AND
childB.type = COLLIDER_LEAF THEN
45                     subCollision ←
CheckOBBTrianglesCollision(childA.obb, childB.vertices,
childB.numVertices / 3)
46
47                 ELSE IF childA.type = COLLIDER_LEAF AND
childB.type = COLLIDER_NODE THEN
48                     subCollision ←
CheckOBBTrianglesCollision(childB.obb, childA.vertices,
childA.numVertices / 3)
49
50                 // both are leaves -> triangle-triangle
collision
51             ELSE
52                 UpdateLeaf(childA, bodyA.transform)
53                 UpdateLeaf(childB, bodyB.transform)
54
55                 (subCollision, manifold) ←
CheckTrianglesCollision(
56                     childA.vertices, childA.numVertices / 3,
57                     childB.vertices, childB.numVertices / 3
58                 )
59
60                 IF subCollision = TRUE THEN
61                     manifold.bodies[0] ← bodyA
62                     manifold.bodies[1] ← bodyB
63                     AddToList(manifolds, manifold)
64                     collision ← TRUE
65                 ENDIF
66             ENDIF

```

```

67
68          // if bounding volumes still overlap, push pair
for deeper testing
69          IF subCollision = TRUE THEN
70              Push(stack, (childA, childB))
71          ENDIF
72      ENDFOR
73  ENDFOR
74 ENDWHILE
75
76 RETURN manifolds
77 ENDFUNCTION

```

Rigid Body Collision Response

This algorithm is responsible for updating the velocities of colliding bodies in order to simulate a collision. It works by firstly checking for collisions between all bodies in the scene and storing the collision manifolds in a list. Then, a stabilisation constant is calculated for each manifold, which is then used in the iterative solving step. In this step, a corrective velocity is added to each body repeatedly based on different constraints – collisions and friction. Over many iterations, the velocities should converge to a global solution which will resolve the collision and other constraints.

```

1 FUNCTION SolveCollisions(bodies, physPeriod)
2     DECLARE contactBuffer ← ARRAY OF Manifold
3
4     // detect all collisions and collect manifolds
5     FOR i ← 0 TO Length(bodies) - 1
6         bodyA ← bodies[i]
7         FOR j ← 0 TO i - 1
8             bodyB ← bodies[j]
9
10        isCollision ← FALSE
11        manifolds ← GetCollisionManifolds(bodyA, bodyB)
12
13        IF manifolds ≠ EMPTY THEN
14            isCollision ← TRUE
15            FOR EACH manifold IN manifolds
16                AddToList(contactBuffer, manifold)
17            ENDFOR
18        ENDIF
19    ENDFOR

```

```

20      ENDFOR
21
22      // precompute stabilisation constants for each manifold
23      FOR EACH manifold IN contactBuffer
24          bodyA ← manifold.bodies[0]
25          bodyB ← manifold.bodies[1]
26
27          manifold.baumgarte ← -BAUMGARTE_BETA / physPeriod *
manifold.penetrationDepth
28
29          restitution ← Min(bodyA.restitution, bodyB.restitution)
30
31          r_ac ← manifold.contactPointA - bodyA.centreOfMass
32          r_bc ← manifold.contactPointB - bodyB.centreOfMass
33
34          deltaV ← bodyB.velocity - bodyA.velocity
35          rwA ← CROSS(bodyA.angularVelocity, r_ac)
36          rwB ← CROSS(bodyB.angularVelocity, r_bc)
37
38          restitutionVector ← deltaV + rwA + rwB
39          restitution ← restitution * DOT(restitutionVector,
manifold.contactNormal)
40          restitution ← Min(0, restitution)
41
42          manifold.baumgarte ← manifold.baumgarte + restitution
43      ENDFOR
44
45      // iterative constraint solving
46      FOR s ← 1 TO NUM_ITERATIONS
47          FOR EACH manifold IN contactBuffer
48              ApplyCollision(manifold.bodies[0],
manifold.bodies[1], manifold)
49          ENDFOR
50      ENDFOR
51  ENDFUNCTION
52
53
54  FUNCTION ApplyCollision(bodyA, bodyB, manifold)
55      invMassMatrix ← SetupInvMassMatrix(bodyA, bodyB)
56      friction ← MAX(bodyA.friction, bodyB.friction)

```

```

57
58     // build constraints for normal and tangents
59     constraints ← BuildConstraints(manifold, bodyA, bodyB)
60
61     FOR c ← 0 TO NUM_CONSTRAINTS - 1
62         // compute denominator
63         lambdaDen ← 0
64         FOR j ← 0 TO 3
65             transformed ← invMassMatrix[j] × constraints[c][j]
66             lambdaDen ← lambdaDen + DOT(constraints[c][j],
transformed)
67         ENDFOR
68
69         // compute numerator
70         lambdaNum ← 0
71         lambdaNum ← lambdaNum - DOT(constraints[c][0],
bodyA.velocity)
72         lambdaNum ← lambdaNum - DOT(constraints[c][1],
bodyA.angularVelocity)
73         lambdaNum ← lambdaNum - DOT(constraints[c][2],
bodyB.velocity)
74         lambdaNum ← lambdaNum - DOT(constraints[c][3],
bodyB.angularVelocity)
75
76         IF c = NORMAL_CONSTRAINT THEN
77             lambdaNum ← lambdaNum - manifold.baumgarte
78         ENDIF
79
80         lambda ← lambdaNum / lambdaDen
81
82         // clamp impulses based on constraint type
83         oldImpulse ← manifold.impulses[c]
84         IF c = NORMAL_CONSTRAINT THEN
85             manifold.impulses[c] ← MAX(oldImpulse + lambda, 0)
86         ELSE
87             limit ← friction *
manifold.impulses[NORMAL_CONSTRAINT]
88             manifold.impulses[c] ← CLAMP(oldImpulse + lambda, -
limit, limit)
89         ENDIF

```

```

90         lambda ← manifold.impulses[c] - oldImpulse
91
92         // apply velocity corrections
93         FOR j ← 0 TO 3
94             deltaV ← (invMassMatrix[j] × constraints[c][j]) *
lambda
95             IF j = 0 THEN bodyA.velocity ← bodyA.velocity +
deltaV
96             IF j = 1 THEN bodyA.angularVelocity ←
bodyA.angularVelocity + deltaV
97             IF j = 2 THEN bodyB.velocity ← bodyB.velocity +
deltaV
98             IF j = 3 THEN bodyB.angularVelocity ←
bodyB.angularVelocity + deltaV
99             ENDFOR
100            ENDFOR
101    ENDFUNCTION

```

TekGL

TekGL is the section of the program that should handle everything related to the rendering of the program. This should include interfacing the window manager, interfacing OpenGL and abstracting both of these into a set of simplified functions. For example, instead of using OpenGL directly to find a free buffer ID, instantiating a vertex buffer at that ID, writing data to the buffer, creating a vertex array and so on, TekGL should make a single function like "tekCreateBuffer()" that hides away this complexity.

Another benefit of TekGL is that, should the program be transferred to a different system such as DirectX, it should be somewhat simpler to just change the implementation of these functions without affecting the rest of the program.

GLFW Interface

TekGL should interface with the GLFW library to manage the creation and lifecycle of windows. GLFW is a library that abstracts the window creation from the operating system and helps to create OS independent applications with OpenGL. There are several key areas that need to be created:

- Creation of a window should be simplified to a single function like "tekInit()" that takes the window position, size and title.
- Responding to user input should be handled using a set of callback functions.
 - There should be functions relating to mouse position, mouse buttons, mouse scrolling and keyboard input.

- There should be a method like “tekAddCallback()” that allows a function to be designated as the callback to one of the input types.
 - Callback functions should be stored in a list to allow multiple functions to be called for each type of input. For example, the game may have a key callback to listen for player movement input, and the GUI may have another key callback to record typing.
 - When a callback is received from GLFW, the callback data should be abstracted to remove information about the internal workings of the window. Then, each callback in the list should be called, passing this data onto it.
- There should be a function that allows for checking if the window has been closed or not. This will allow for a loop to be created that reads like “Repeat while the window should not be closed”.
- There should be a function that allows the mouse cursor image to be changed when needed.
- Deleting a window should be simplified to another function that cleans up the window and any supporting data structures such as the callback lists.

OpenGL / GLAD Interface

TekGL also needs to integrate with OpenGL, which requires the use of a library like GLAD. OpenGL is only a specification of functions, and each operating system will have its own implementation of OpenGL. The GLAD library is responsible for finding this implementation, and providing access to the function pointers that have been provided by that OpenGL implementation.

Meshes

TekGL should be capable of managing meshes, these are a simplified view of the actual data that is required to render an object. TekGL should be able to:

- Creating a mesh given a set of vertices, indices and the arrangement of the vertices. These are the same properties as seen in the model file / model loader from TekPhys. This is done by creating a array buffer for the vertices, an element array buffer for the indices, and a vertex array to link these all together.
- Drawing a mesh to the screen provided that a mesh has been previously created.
- Deleting a mesh, freeing any allocated memory and telling OpenGL to delete the buffers as well.

Shaders

TekGL should also be able to load shaders that are written in GLSL (OpenGL shader language). Shaders are programs that determine how to colour the screen based on the objects that are inputted into them. They are designed to

be reused by many objects, so all objects drawn are likely to go through the same shader. Shaders can come in one of three varieties, a vertex shader, a geometry shader and a fragment shader. A vertex shader transforms the points of a mesh onto the screen. The geometry shader takes the transformed points and decides how to connect the points into geometry such as triangles that can be drawn. The geometry shader can be omitted if drawing simple triangles or lines as OpenGL has options for these by default. The fragment shader is responsible for colouring in the final geometry, this is where lighting calculations happen and such. In order to read a shader program, TekGL will need to:

1. Read each shader file from the filesystem into an array of characters.
There should be at least a vertex and fragment shader, and optionally a geometry shader.
2. For each file, pass the data to OpenGL so it can compile the shader into machine code. Check for compilation errors at this stage.
3. Now link the shader programs together

Once the shader is compiled, TekGL will also need functionality to delete shaders, as well as setting shader uniforms – these are variables that can be externally modified. They are often used to store data that remains constant across different objects such as the position and colour of lights in the scene.

Camera

The camera is an important part of 3D graphics. TekGL should create a simple structure that stores information about the camera, this includes the position and rotation, and the associated matrices that allow for the scene to be transformed to what the camera should be able to “see”. Storing this information all together makes it much simpler to pass into the shader where it is actually needed for rendering. The cglm library provides functionality for converting the camera position and rotation into a matrix that can be used by the shaders.

Textures

Textures are simply images that can be drawn to the screen. You can imagine it like wallpaper, a pattern or image that can be attached onto objects, but instead of glueing it onto a wall, a shader program will automatically draw it on there for you. A texture is stored on your computer as a normal image file, such as a .jpeg, a .png or a .bmp file. TekGL should use a library called stb_image.h (or another library could be used for different languages than C or C++), which allows for images to be converted from a .jpeg or a .png into the actual raw data for colours per pixel. Essentially, it is an image decompressor. This is because OpenGL expects to receive the raw colour data for images and not the compressed version. There are some optional settings such as how should the texture repeat, and whether mipmaps be used to optimise performance.

Once created, textures can be bound to a texture slot, this allows for shaders to access the texture using this slot ID and use it when rendering. This will be done in the fragment shader.

Text / Font

Text is a crucial part of any program, as it is often needed to display information to the user that images and objects cannot. Text is not possible to draw by default using OpenGL, so therefore it is required to create a text renderer. To display text, we create a mesh of rectangles, where each rectangle is the bounding box of a single character. Each vertex will contain texture coordinate data that specifies where to read a character from a font atlas, which is a large texture containing one of each possible character.

TekGL will need to read font files (.ttf / truetype font) from a file, and create the font atlas texture from this. This can be achieved using the FreeType library, and using it to retrieve the bitmap image of each character (also called a glyph), as well as its size, position, height over the baseline, spacing between characters, and then adding each one to a larger texture that can contain all of the characters.

The text renderer itself has two main functions, which are to create a mesh containing the text data, and to draw the mesh with the correct shaders and textures to display it. In order to create a text mesh, use the following algorithm:

```
1 SUBROUTINE GenerateTextMeshData(text, len_text, size, font,
text_obj, OUT vertices, OUT len_vertices, OUT indices, OUT
len_indices)
2     # initialise text dimensions
3     text_obj.width ← 0
4     text_obj.height ← 0
5
6     DECLARE vertices ← ARRAY OF vertex
7     DECLARE indices ← ARRAY OF integer
8
9     # expose lengths to caller
10    len_vertices ← len_vertices_local
11    len_indices ← len_indices_local
12
13    # drawing pointer (top-left baseline)
14    x ← 0.0
15    y ← FLOAT(size)
16
17    # precalculated scale values
```

```

18     scale ← FLOAT(size) / FLOAT(font.original_size)
19     atlas_size ← FLOAT(font.atlas_size)
20
21     # for each character in the string
22     FOR i ← 0 TO len_text - 1
23         ch ← text[i]
24
25         # handle newline: move to next line
26         IF ch = '\n' THEN
27             IF x > text_obj.width THEN
28                 text_obj.width ← x
29             ENDIF
30             x ← 0.0
31             y ← y + FLOAT(size)
32             text_obj.height ← text_obj.height + FLOAT(size)
33             CONTINUE FOR
34         ENDIF
35
36         # fetch glyph for this character (glyph contains width,
height, bearing_x, bearing_y, atlas_x, atlas_y, advance)
37         glyph ← font.glyphs[ASCII(ch)]
38
39         glyph_width ← FLOAT(glyph.width)
40         glyph_height ← FLOAT(glyph.height)
41
42         # top-left position for this glyph in screen space
43         x_pos ← x + FLOAT(glyph.bearing_x) * scale
44         y_pos ← y + (glyph_height - FLOAT(glyph.bearing_y)) *
scale
45
46         # screen size of the glyph (height negative if y-axis
inverted for rendering)
47         width ← glyph_width * scale
48         height ← -glyph_height * scale
49
50         # atlas pixel coordinates
51         atlas_x ← FLOAT(glyph.atlas_x)
52         atlas_y ← FLOAT(glyph.atlas_y)
53
54         # texture coordinates (u,v) in range [0,1]

```

```

55     tex_u0 ← atlas_x / atlas_size
56     tex_v0 ← atlas_y / atlas_size
57     tex_u1 ← (atlas_x + glyph_width) / atlas_size
58     tex_v1 ← (atlas_y + glyph_height) / atlas_size
59
60     # write vertex data for this glyph (4 vertices × [x, y,
u, v])
61     vert_offset ← i * 16
62     vertices[vert_offset + 0] ← x_pos
63     vertices[vert_offset + 1] ← y_pos + height
64     vertices[vert_offset + 2] ← tex_u0
65     vertices[vert_offset + 3] ← tex_v0
66
67     vertices[vert_offset + 4] ← x_pos
68     vertices[vert_offset + 5] ← y_pos
69     vertices[vert_offset + 6] ← tex_u0
70     vertices[vert_offset + 7] ← tex_v1
71
72     vertices[vert_offset + 8] ← x_pos + width
73     vertices[vert_offset + 9] ← y_pos
74     vertices[vert_offset + 10] ← tex_u1
75     vertices[vert_offset + 11] ← tex_v1
76
77     vertices[vert_offset + 12] ← x_pos + width
78     vertices[vert_offset + 13] ← y_pos + height
79     vertices[vert_offset + 14] ← tex_u1
80     vertices[vert_offset + 15] ← tex_v0
81
82     # write index data for two triangles (using 4 vertices
per glyph)
83     idx_offset ← i * 6
84     base_index ← i * 4
85
86     indices[idx_offset + 0] ← base_index + 0
87     indices[idx_offset + 1] ← base_index + 1
88     indices[idx_offset + 2] ← base_index + 2
89     indices[idx_offset + 3] ← base_index + 0
90     indices[idx_offset + 4] ← base_index + 2
91     indices[idx_offset + 5] ← base_index + 3
92

```

```

93          # advance drawing pointer horizontally by glyph's
advance (advance is in 1/64 pixels in many font formats)
94          x ← x + FLOAT(glyph.advance >> 6) * scale
95          NEXT i
96
97          # final width/height adjustments for last line
98          IF x > text_obj.width THEN
99              text_obj.width ← x
100         ENDIF
101         text_obj.height ← text_obj.height + FLOAT(size)
102
103         # return results to caller
104         vertices ← vertices
105         indices ← indices
106
107         RETURN TRUE
108 ENDROUTINE

```

Materials

In TekGL, a material should be an abstract collection of a shader program and shader data. It should be stored in a .yml file that is read to retrieve the properties of this material. Materials are a purely visual effect, and do not affect the physical properties of the material they are drawn on. A material should contain two sections, the “shaders” section and the “uniforms” section.

The shaders section should contain the filename of the vertex and fragment shaders, and optionally the geometry shader.

The uniforms section should have a set of keys and values, where the key correlates to the name of a shader uniform’s name in one of the shaders specified. The value is what should be inputted into the shader. Any uniforms that have a string as their value are assumed to be a texture uniform, and in this case the string is interpreted as the filename of a texture which is loaded and assigned to a slot, and this slot is passed as the uniform instead of the string. A uniform can also have a vector value by specifying values for either x, y, z and optionally w, or r, g, b, and optionally a. To specify variable data to use in the shader uniforms, there are four wildcard strings that can be set as uniform values that will not be interpreted as an image file. These are “\$tek_camera_position”, “\$tek_model_matrix”, “\$tek_view_matrix”, and “\$tek_projection_matrix”. When these values are passed in, the value of the uniform will be set to the camera’s position, the model matrix, the view matrix or the projection matrix respectively.

The material is stored as an array of uniform structures. Each uniform structure should have a uniform type that represents the type, either a number, a vector,

a matrix or a texture ID. The structure should also contain the data that should be passed into the shader.

A typical material file may look something like this:

```
shaders:
    vertex_shader: "../shader/vertex.glvs"
    fragment_shader: "../shader/fragment.glfs"
uniforms:
    light_color:
        r: 0.9
        g: 0.9
        b: 0.9
    light_position:
        x: 4.0
        y: 12.0
        z: 0.0
    camera_pos: $tek_camera_position
    model: $tek_model_matrix
    view: $tek_view_matrix
    projection: $tek_projection_matrix
```

Entities

In TekGL, an entity should simply be a data structure that contains a mesh, a material, and a position, rotation and scale. This allows all of the data to draw a single object to be kept in the same place, without having the need to have separate lists for meshes, materials, positions and so on. An entity should also have an associated function to draw itself, this will sequentially bind the shader specified by the material, set the uniforms of the material, bind the mesh, and call the draw function of the mesh to display it to the screen.

TekGui

TekGui is an extension of TekGL that caters towards rendering different GUI components to the user. These are designed to be modular and reusable across different parts of the GUI, as stand-alone parts or incorporated into larger GUI structures.

All GUI structures should have the ability to be tweaked using .yml files. For example, most GUI structures should have a foreground and background colour – these should be set in the configuration files. This allows for quick changes to the GUI if there is ever a decision to change the colour scheme of the application, or to change the size and position of different GUI elements.

Button

In TekGui, a button is simply an area of the window that should respond to mouse inputs. It should have a bounding box specified by a position, a width and a height. It should also have a callback that can be changed to run a user-defined function when mouse input is received in the bounds of the button.

In order to detect if a button is being clicked, we calculate the minimal and maximal x and y coordinates of the button. We also set up a mouse move and mouse click callback using the functionality from TekGL. For each mouse event, compare the cursor coordinates to the minimal and maximal button coordinates. If $\text{min_x} < \text{x} < \text{max_x}$ and $\text{min_y} < \text{y} < \text{max_y}$, then it means that the mouse is inside the button.

It is possible that buttons could overlap each other, causing confusion on which one has been clicked. In TekGL, each button that is created should be added to a list of buttons, this represents the layering of buttons relative to one another. When a mouse input is received, iterate over the buttons in this list, and as soon as one button registers a mouse event within its bounds, the loop should exit leaving only one button to register the click.

TekGL should implement functionality to change the ordering of a button, by moving it to the front of the list to change its priority. This will correlate to actions like selecting a window and moving it on top of another window – now the buttons on this window should be moved to the front.

Text Button

A text button should inherit from the features of a button, but should also draw the bounds of the button onto the screen. When the mouse hovers over the button, the button should change colour to show that it is selected. The text button should also include a text mesh that contains a label for this button, which should be drawn in the centre of the button.

The text button should have a section in the configuration file to change the default position, size, colour, hover colour, border colour, border width and text size.

Text Input

A text input should be a single line of editable text. It will incorporate a button to listen for when the user clicks on the text input, this will signal that the text needs to be edited. When editing, a cursor should appear and flicker on the position in the input where new characters will be inserted or removed. The input should listen for key inputs, and add the appropriate character to the text input when it is typed. If the return key is pressed, the input should stop listening and the cursor should disappear. The left and right arrow keys should be able to move the cursor in their respective direction so long as the start or the end of the input is not surpassed by the cursor. If the length of the message typed is larger than the width of the text input, then the left and right arrow

keys should become able to control the position of the cursor to adjust which part of the text is being displayed.

Window

TekGui will provide access to basic window functionality, where a window has a position, width, height, background colour, border colour, and a title. The background colour should have the possibility of being transparent. Similar to buttons, windows should all be stored in a list that represents which windows are displayed above others. However, due to the way rendering works, the first window in the list will visually be at the bottom of other windows, as each window drawn after the first one will be drawn on top of it. If a window is to be brought forwards, then it should move to the end of the list instead of the start. The window should include a button (not a text button) that is the same shape as the top bar of the window. It should listen to mouse events, and if the mouse is held while inside this bar then the window should move by the same amount as the mouse – this allows users to drag windows around to where they would like. Windows should also have a draw callback that is called every time the window is drawn. This will allow for windows to be extended to have extra functionality and options.

More complicated windows will all inherit from the base window. This means that windows will maintain a consistent style, and the style of the GUI can be easily changed by editing the configuration file.

List Window

A list window will be able to draw a list of strings. It will have a large button that covers the area that the list of strings will take up. The list window will listen to callbacks for this button, and interpolate for which item is being clicked based on the number of items being displayed and the height of the button. The window should have an editable callback that will provide the index of the item being clicked instead of the raw mouse data. The window should also have a hash table with string keys and values of text meshes. This means that if multiple items in the list represent the same string, they can all share a single text mesh and avoid using more memory than is required to store all of the strings in the list.

Option Window

An option window will be able to draw a list of “options” – these are different kinds of input such as strings, numbers, vectors, booleans and buttons. An option window should be created using a .yml file, which contains the window position, size and other metadata, as well as the list of options to be displayed. Each option should have a name which is used to access its value in code. Each should also have a label which is some text to display related to this option, a type which specifies how many inputs, what type of input and how to validate it, e.g. a type of “vec3” will have 3 text inputs which should be validated to ensure they are numbers. Each item should also have an index, which specifies the

order in which items are drawn. Internally, when each item is loaded from the file it should be added to a priority queue, where the priority is the index. Then, dequeuing this queue will give the final order of the options. This means that when options are added, they can be given indices like 10, 20 and 30, and if a new option is added in between it could have an index of 25 to avoid having to edit all the existing options, while still placing it between the options. Once created, options are stored in a hash table with a string key that is the same as the name specified in the .yml configuration file, and the value will be the data related to the option. Alongside this, there is an array of “display options” which will contain the name of the option they correspond to, as well as the physical GUI components that are needed to draw them. Additionally, they require a height value that tells the next option how far down to move before it starts drawing, this avoid options being drawn on top of each other.

As the “option” and “display option” are very similar, a table is provided below to fully explain the difference between them and the reason for each one’s existence.

	Option	Display Option
Location in the option window.	Stored in a hash table, where the key is the name of the option.	Stored in an array, where the order of the array represents the drawing order.
Purpose	To store the data that is kept by each option.	To store the GUI components that allow the user to interact with the window.
Required attributes	<ul style="list-style-type: none"> • Type • One of: <ul style="list-style-type: none"> ◦ String ◦ Number ◦ Boolean ◦ 3 component vector ◦ 4 component vector 	<ul style="list-style-type: none"> • Type • Height • One of: <ul style="list-style-type: none"> ◦ Text ◦ Text input, Option name and Index (if multiple inputs correspond to the same name in the case of a vector) ◦ Text button and Option name
Can it be read or written to?	Can be read or written to at any point by providing the name of the key, and using the appropriate function for the type	Cannot be edited once the window is created, only based on what is contained in the .yml file.

	of data.	
Can it be seen on the screen?	No, the values can only be read in code. These values are read by the options which are then updated to show changes on the screen.	Yes, these are what are displayed on the screen.

File Structures

Project Structure

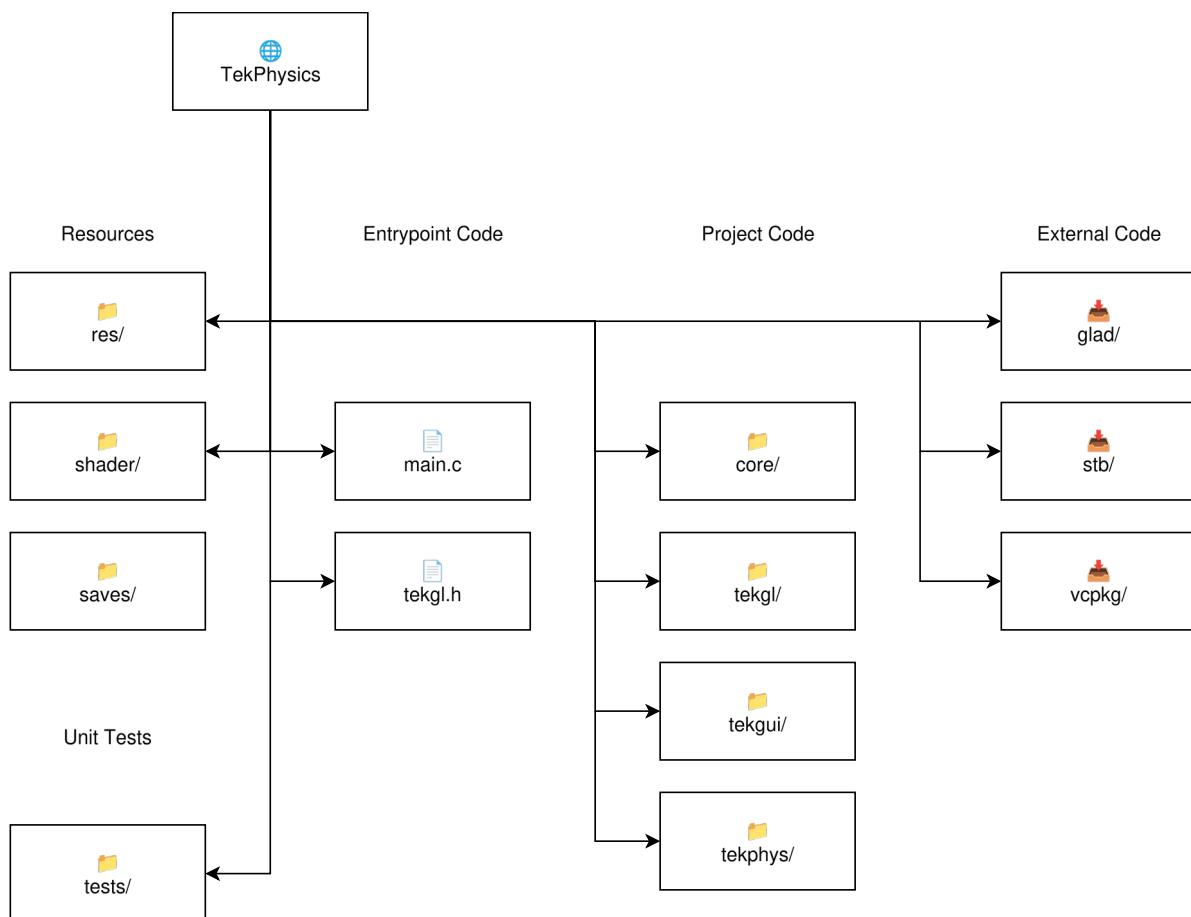


Figure 27: The main structure of the project.

- res/ - Folder containing different resources such as textures and meshes for the simulator.
- shader/ - Folder containing different shader programs that the simulator will use when rendering.
- saves/ - Folder containing the user's saved data for scenarios they have made.

- tests/ - Folder containing different unit tests to ensure that the program is working correctly.
- core/ - Folder containing various utilities that are used throughout the program source code.
- tekgl/ - Folder containing various graphics functionality and OpenGL abstractions.
- tekgui/ - Folder containing various GUI components and controls.
- tekphys/ - Folder containing various physics simulation algorithms and utilities.
- main.c - The entry point of the application, containing different startup procedures and the main rendering loop.
- tekgl.h - A generic header for common utilities used throughout the source code.
- glad/ - Source code of the GLAD package, which is required for OpenGL to work.
- stb/ - The stb image library, required for loading compressed images like .png and .jpg files.
- vcpkg/ - The vcpkg manager, contains other libraries that were imported into the project.

Core Folder Structure

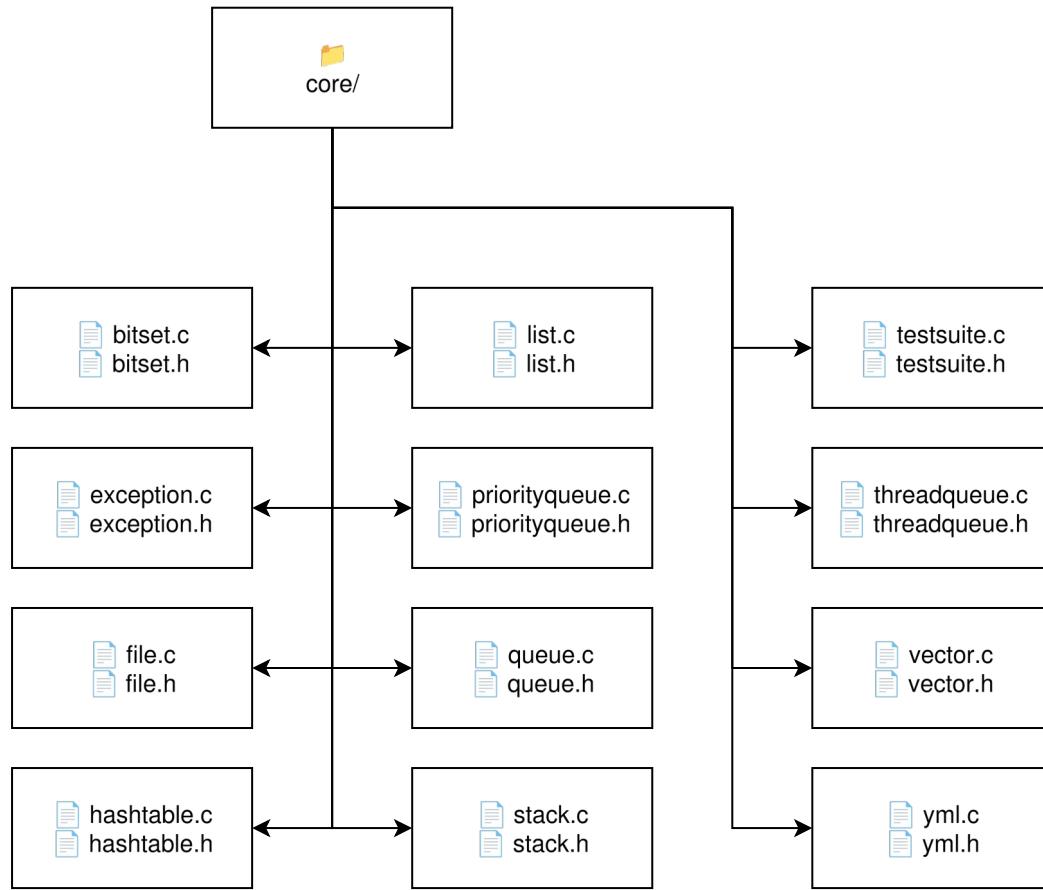


Figure 28: The structure of the "core" folder.

- `bitset.c/bitset.h` – Function suite providing functionality for a bit set, allows the compact storage of a set of boolean values.
- `exception.c/exception.h` – Function suite providing exception handling and stack tracing.
- `file.c/file.h` – Function suite providing access to the filesystem.
- `hashtable.c/hashtable.h` – A hash table implementation allowing for an unordered, fast access, key-value mapping between different data.
- `list.c/list.h` – A linked list implementation providing functionality for adding, removing, inserting and moving items in the list.
- `priority.c/priorityqueue.h` – A priority queue implementation that resembles a linked list.
- `queue.c/queue.h` – A queue implementation that internally resembles a linked list.
- `stack.c/stack.h` – A stack implementation that internally resembles a linked list.
- `testsuite.c/testsuite.h` – Suite of functions to simplify unit testing.

- `threadqueue.c/threadqueue.h` – A thread-safe one-way queue implementation, uses a circular queue philosophy, fixed buffer size and atomic queue and dequeue pointers.
- `vector.c/vector.h` – A vector implementation (resizing array) that doubles the length of the available buffer when an item is appended that cannot fit the current size.
- `yml.c/yml.h` – Allows YAML files to be read and converted into a data structure that can be easily accessed.

TekGL Folder Structure

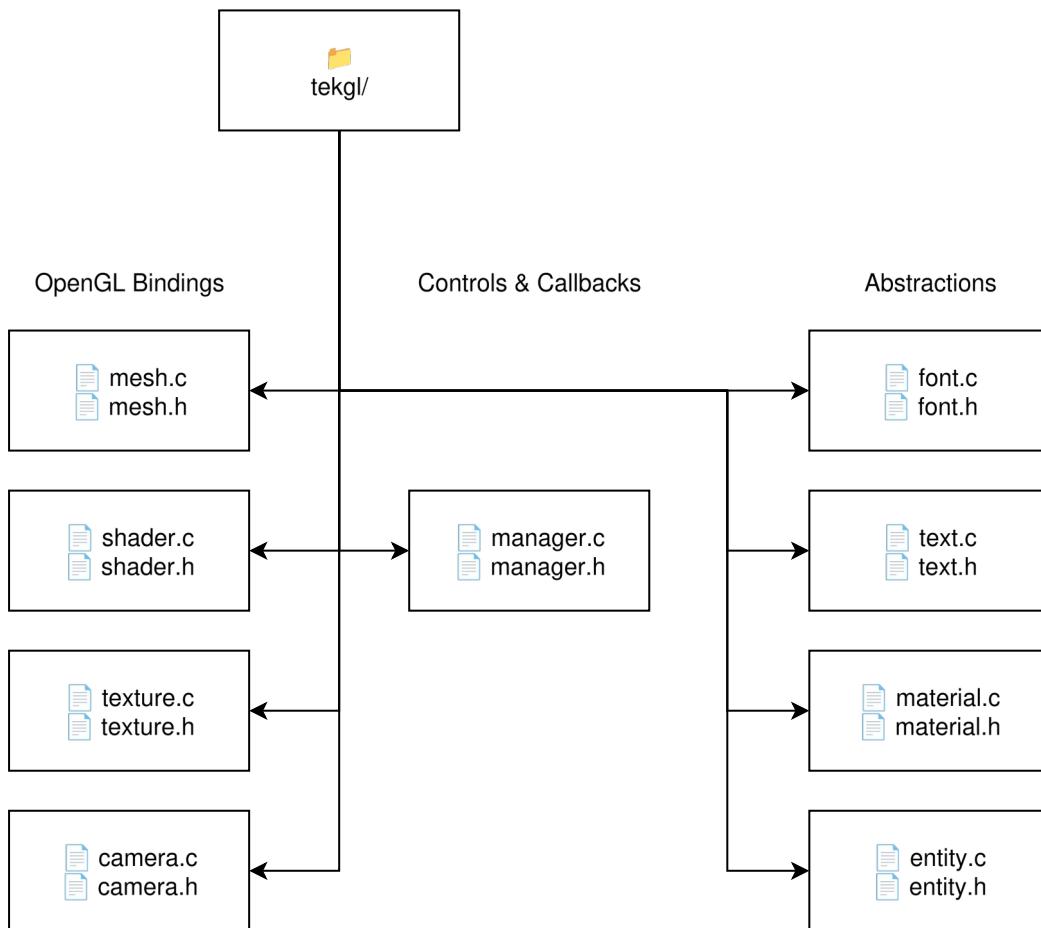


Figure 29: The structure of the "tekgl" folder.

- `mesh.c/mesh.h` – Responsible for reading mesh files, and interfaces with OpenGL to store read data.
- `shader.c/shader.h` – Responsible for reading, compiling and using shader program source code.
- `texture.c/texture.h` – Responsible for reading and loading images as textures.
- `camera.c/camera.h` – Responsible for storing camera position and rotation, and has other functions for camera controls.

- manager.c/manager.h – Allows other parts of the program to interface with different graphics and window functions.
- font.c/font.h – Responsible for loading fonts used to render text.
- text.c/text.h – Responsible for generating meshes that can be rendered to show text.
- material.c/material.h – An abstraction that allows a YAML file to be read that includes shader and texture data, automatically binding everything in OpenGL.
- entity.c/entity.h – Responsible for maintaining a relationship between a mesh, material, position and rotation as a complete entity.

TekPhys Folder Structure

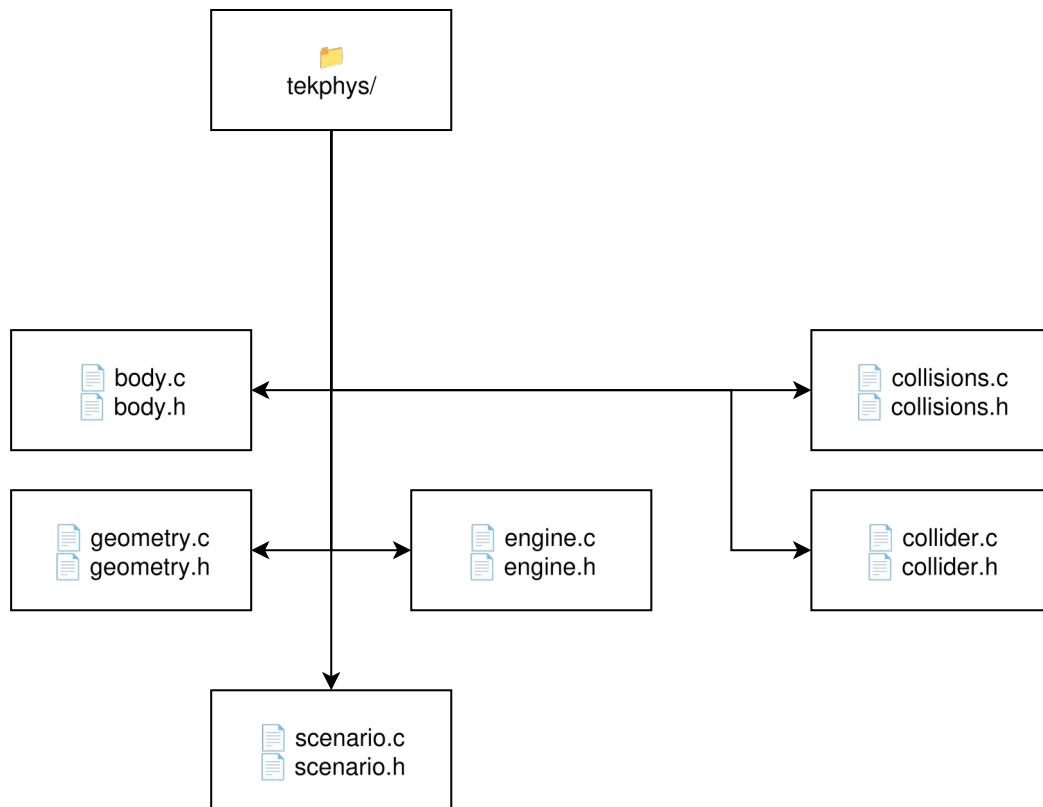


Figure 30: The structure of the "tekphys" folder.

- body.c/body.h – A set of functions that simulates a rigid body, storing its position, rotation, mass, inertia tensor and collider.
- geometry.c/geometry.h – A set of utility functions for different mathematical and geometric calculations.
- engine.c/engine.h – Contains the functionality for the main physics thread and supporting code.
- collisions.c/collisions.h – A set of functions related to testing for collisions and providing the appropriate collision response.

- `collider.c/collider.h` – A set of functions related to generating a collision structure from a mesh.

Res Folder Structure

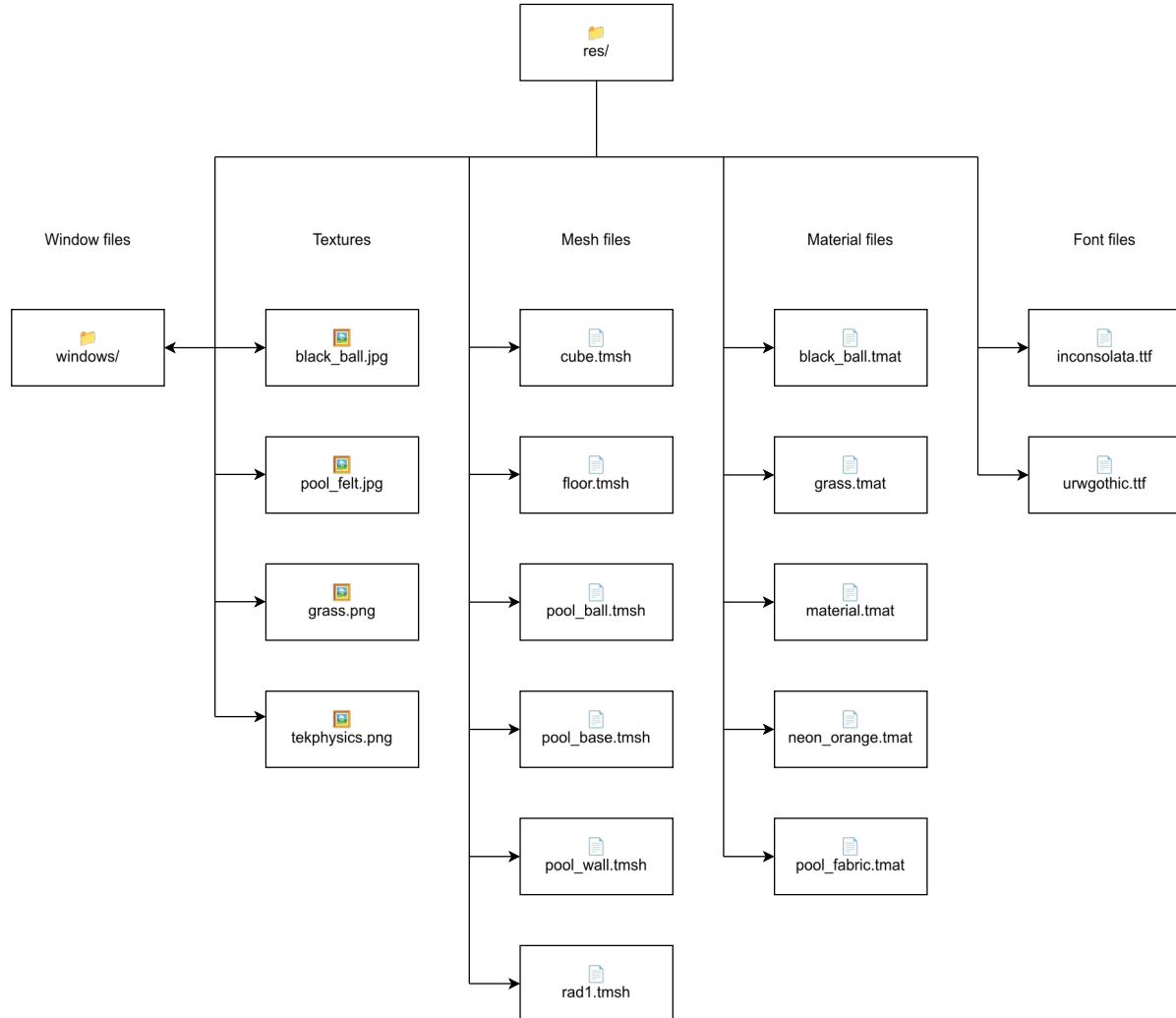


Figure 31: The resources folder structure.

- `windows/` - A folder containing the different windows that are rendered in the GUI.
- `black_ball.jpg` – A texture of a black '8' ball in pool.
- `pool_felt.jpg` – A texture of green felt that is used on pool tables.
- `grass.png` – A texture of grass that is found growing outside on the ground.
- `tekphysics.png` – A texture of the TekPhysics logo.
- `cube.tmsh` – A model file containing the data of a cube.
- `floor.tmsh` – A model file containing the data of a flat plane / floor.
- `pool_ball.tmsh` – A model file containing the vertices of a small isosphere.

- pool_base.tmsh – A model file containing the vertices of a small slab.
- pool_wall.tmsh – A model file containing the vertices of a cushion of a pool table.
- rad1.tmsh – A model file containing the vertices of an isosphere with a one metre radius.
- black_ball.tmat – A material file containing the material for a black '8' ball in pool.
- grass.tmat – A material file containing the material for grass.
- material.tmat – A generic material with a grey colour.
- neon_orange.tmat – A bright orange coloured material.
- pool_fabric.tmat – A material of the fabric on pool tables.
- inconsolata.ttf – A TrueType font file for a monospaced font named Inconsolata.
- urwgothic.ttf – A TrueType font file for a font named URW Gothic.

Windows Folder Structure

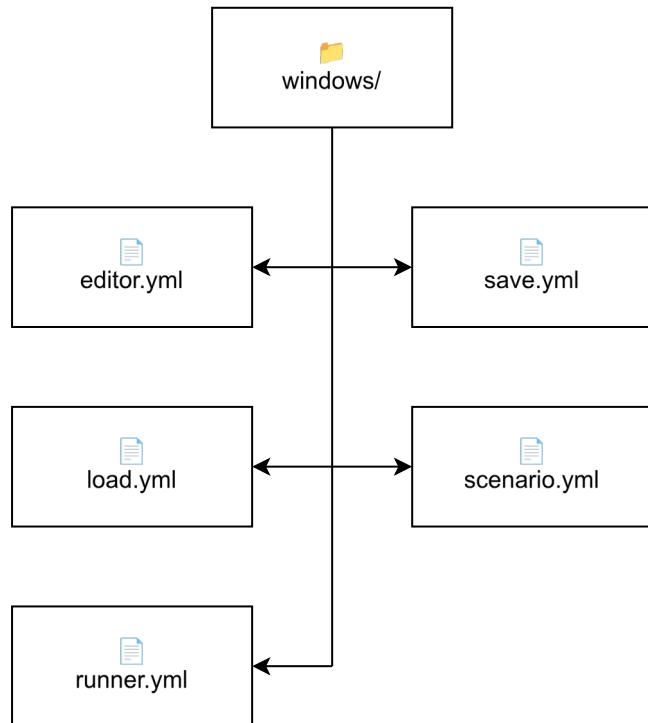


Figure 32: A diagram showing the structure of the "windows" folder.

- editor.yml – A file containing the information needed to draw the editor window.

- load.yml – A file containing the information needed to draw the load window.
- runner.yml – A file containing the information needed to draw the runner window.
- save.yml – A file containing the information needed to draw the save window.
- scenario.yml – A file containing the information needed to draw the scenario window.

Shader Folder Structure

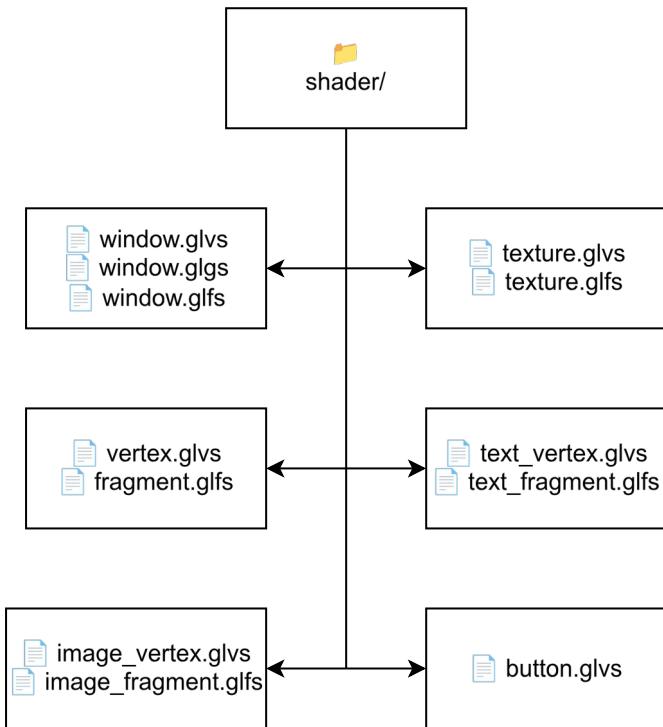


Figure 33: A diagram showing the structure of the "shader" folder.

- window.(glvs/glgs/glfs) – Shader for drawing windows.
- vertex.glvs / fragment.glfs – Default shader for drawing simple materials.
- image_vertex.(glvs/glfs) – Shader for drawing 2D images from textures.
- texture.(glvs/glfs) – Shader for drawing 3D objects with texture.
- text_(vertex.glvs/fragment.glfs) – Shader for drawing text.
- button.glvs – Slightly different vertex shader for buttons, but borrows window.glgs and window.glfs to make a full shader.

Tests Folder Structure

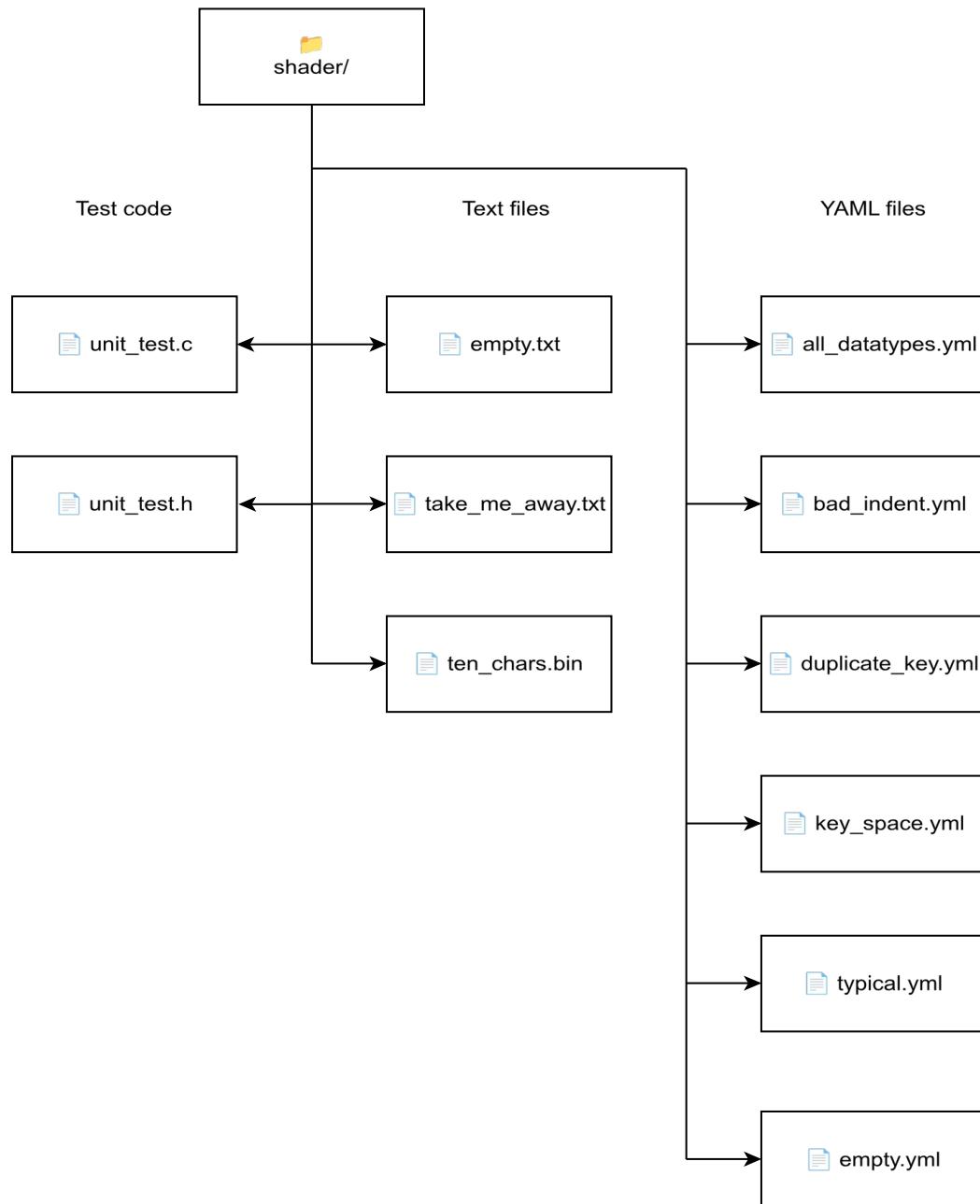


Figure 34: A diagram showing the structure of the "tests" folder.

- `unit_test.c/h` – The code that contains all of the unit tests
- `empty.txt` – A completely empty text file to test what happens.
- `take_me_away.txt` – A text file containing some of the lyrics for the song "Take Me Away – True Faith with Final Cut (Pin Up Girls Remix)".
- `ten_chars.bin` – A text file containing exactly ten characters.
- `all_datatypes.yml` – A yml file containing a key for each possible data type.
- `bad_indent.yml` – A yml file containing a bad indent.

- `duplicate_key.yml` – A yml file containing a duplicated key.
- `key_space.yml` – A yml file that has a key with a space in it.
- `typical.yml` – A yml file being used for a typical usage.
- `empty.yml` – A yml file that is completely empty.

Function Listing

TekGL

Name	Parameters	Return	Description
<code>EXPAND_VEC3</code>	N/A	N/A	Take a 3 component vector and expand it, used for printing mostly.
<code>EXPAND_VEC4</code>	N/A	N/A	Take a 4 component vector and expand it, used for printing mostly.

Main

Name	Parameters	Return	Description
<code>tekPushInspectEvent</code>	<code>inspect_id: uint</code>	N/A	Push an inspect event to the event queue. Changes which body is shown to the display.
<code>tekMainKeyCallback</code>	<code>key: int, scancode: int, action: int, mods: int</code>	N/A	The main key callback of the program. Listens for the W, A, S and D keys to allow for camera to move.
<code>tekMainMousePositionCallback</code>	<code>x: double, y: double</code>	N/A	The main mouse position callback of the program. Used to update the camera rotation.
<code>tekMainMouseButtonCallback</code>	<code>button: int, action: int, mods: int</code>	N/A	The main mouse button callback of the program. Used to track if the right mouse button is pressed, which would determine if the camera should rotate or not.
<code>tekBodyCreateEvent</code>	<code>snapshot: TekBodySnapshot*, snapshot_id: int</code>	N/A	Push a body create event to the event queue. NOTE: The body snapshot is copied into the event.
<code>tekCreateBodySnapshot</code>	<code>scenario: TekScenario*, snapshot_id: int*</code>	N/A	Create a body snapshot filled with default values and add it to a scenario. Also chooses a new ID for the created body.

tekUpdateBodySnapshot	scenario: TekScenario*, snapshot_id: int	N/A	Push a body update event to the event queue. Uses the body snapshot data stored at the specified ID in the scenario.
tekDeleteBodySnapshot	scenario: TekScenario*, snapshot_id: int	N/A	Delete a body snapshot from a scenario, and push a body delete event to the event queue.
tekRecreateBodySnapshot	snapshot: TekBodySnapshot*, snapshot_id: int	N/A	Recreate a body from a body snapshot. This should be called if a large change happens to a body, such as changing the model so that a new collision structure can be made. NOTE: Functionally this is the same as deleting the body and making a new one with the same ID.
tekChangeBodySnapshotModel	scenario: TekScenario*, snapshot_id: int, model: char*	N/A	Change the model of a snapshot body. This needs a separate function as changing a model requires the physics body to be recreated.
tekChangeBodySnapshotMaterial	scenario: TekScenario*, snapshot_id: int, material: char*	N/A	Change the material of a snapshot body. This requires a separate function because the buffer containing the material filename needs to be reallocated.
tekResetScenario	scenario: TekScenario*	N/A	Reset the scenario so that there are no bodies left, and it is back to its original state.
tekRestartScenario	scenario: TekScenario*	N/A	Restart a scenario, updating all bodies to the values specified by the scenario.
tekSimulationSpeedEvent	rate: double, speed: double, runner_window: TekGuiOptionWindow*	N/A	Push a time event to the event queue, which will change the rate and speed of the simulation.
tekGravityEvent	gravity: double, runner_window: TekGuiOptionWindow*	N/A	Push a gravity event to the event queue, which will change the acceleration due to gravity.

tekPauseEvent	paused: flag	N/A	Push a pause event to the event queue, which will stop the simulation temporarily.
tekHideAllWindows	gui: TekGuiComponents*	N/A	Hide all the possible windows from each menu mode.
tekSwitchToMainMenu	gui: TekGuiComponents*	N/A	Switches the menu mode into the main menu mode. Brings the start button to the front.
tekSwitchToBuilderMenu	gui: TekGuiComponents*	N/A	Switches the menu mode into the builder menu mode. Makes all relevant windows visible and brings them to the front.
tekSwitchToSaveMenu	gui: TekGuiComponents*	N/A	Switches the menu mode to the save menu mode. Brings the save window to the front and re-centres it.
tekSwitchToLoadMenu	gui: TekGuiComponents*	N/A	Switches the menu mode to the load menu mode. Brings the load window to the front and re-centres it.
tekSwitchToRunnerMenu	gui: TekGuiComponents*	N/A	Switches the menu mode to the runner menu mode. Displays the runner window and brings it to the front.
tekChangeMenuMode	gui: TekGuiComponents*	N/A	Switches the menu mode based on what the next menu mode is specified to be. Resets the next mode to -1. NOTE: Uses the static variables mode and next_mode, this allows different callbacks to edit the mode.
tekStartButtonCallback	button: TekGuiTextButton*, callback_data: TekGuiButtonCallbackData	N/A	The callback for the start button. If the button was left-clicked, the mode is changed to builder.
tekCreateMainMenu	window_width: int, window_height:	N/A	Create the main menu, this includes creating the logo, the start button and the splash text.

	int, gui: TekGuiComponents*		
tekDrawMainMenu	gui: TekGuiComponents*	N/A	Draw the main menu. Draws the logo, start button and splash text.
tekDeleteMainMenu	gui: TekGuiComponents*	N/A	Delete the main menu, clearing any memory from the gui components used.
tekUpdateEditorWindow	editor_window: TekGuiOptionWindow*, scenario: TekScenario*	N/A	Update the editor window with the current values of a body snapshot. NOTE: Uses the currently selected object in the hierarchy window as the ID.
tekHierarchyCallback	hierarchy_window: TekGuiListWindow*	N/A	The callback for any updates to the hierarchy window. Updates the currently selected body snapshot.
tekDisplayOptionError	window: TekGuiOptionWindow*, key: char*, error_message: char*	N/A	Set a string input to display an error.
tekReadNewFileInput	window: TekGuiOptionWindow*, key: char*, directory: char*, extension: char*, filepath: char**	N/A	Read a filename for a new, unwritten file. Checks to make sure the input is not empty, but will allow any filename otherwise. If the inputted filename ends with the requested extension, it is not appended. NOTE: "filepath" is allocated, so needs to be freed by the caller.
tekReadExistingFileInput	window: TekGuiOptionWindow*, key: char*, directory: char*, extension: char*, filepath: char**	N/A	Get the name of an existing file. The function will reject any inputs if the file specified does not exist, or if the input string is empty. If a filename exists with the same name but the real file includes the specified extension, then this input is accepted. NOTE: The filepath is allocated, and needs to be freed

			by the caller.
tekEditorCallback	window: TekGuiOptionWindow*, callback_data: TekGuiOptionWindowCallbackData	N/A	The callback for the editor window. This is called whenever a user updates a body snapshot's properties or deletes it.
tekActionCallback	window: TekGuiListWindow*	N/A	The callback for the action window. Called whenever the user selects an option like "save", "load" "run" or "quit".
tekCreateActionsList	actions_list: List**	N/A	Create the list of possible actions for the action window.
tekScenarioCallback	window: TekGuiOptionWindow*, callback_data: TekGuiOptionWindowCallbackData	N/A	The callback for the scenario options window. Called when options such as gravity, sky colour and start paused are changed.
tekCreateBuilderMenu	window_width: int, window_height: int, gui: TekGuiComponents*	N/A	Create the builder menu, this includes creating all the windows related to the object hierarchy, the options menu, the editor window and the scenario window.
tekDrawBuilderMenu	gui: TekGuiComponents*	N/A	Draw the builder menu. Draws all visible windows, and hides the editor window if the hierarchy index is less than 0.
tekDeleteBuilderMenu	gui: TekGuiComponents*	N/A	Delete the builder menu, freeing any memory that was allocated by its gui components.
tekSaveCallback	window: TekGuiOptionWindow*, callback_data: TekGuiOptionWindowCallbackData	N/A	The callback for the save window. Called when the user updates the save location or presses the save button.

tekLoadCallback	window: TekGuiOptionWindow*, callback_data: TekGuiOptionWindowCallbackData	N/A	The callback for the load window. Called when the user updates the load location or presses the load button.
tekRunnerCallback	window: TekGuiOptionWindow*, callback_data: TekGuiOptionWindowCallbackData	N/A	The callback for the runner window, called when user updates the speed or rate of the simulation, or when they press play, pause, stop or step.
tekInspectDrawCallback	window: TekGuiWindow*	N/A	The window draw callback for the inspect window. Called every frame that the window is drawn.
tekWriteInspectText	string: char*, max_length: size_t, time: float, fps: float, name: char*, position: vec3, velocity: vec3	The number of characters that could not be written because they did not fit in the buffer: int	Write the inspector text, wrapper around a call to sprintf that has the format string.
tekUpdateInspectText	inspect_text: TekText*, time: float, fps: float, name: char*, position: vec3, velocity: vec3	N/A	Update the inspection text with new information that is useful to the user.
tekCreateRunnerMenu	gui: TekGuiComponents*	N/A	Create the runner menu, which has options related to controlling the running of the scenario such as speed and time controls.
tekDrawRunnerMenu	gui: TekGuiComponents*	N/A	Draw the runner menu to the screen.
tekDeleteRunnerMenu	gui: TekGuiComponents*	N/A	Delete the runner memory, freeing any allocated memory for this section of the gui.
tekGetSplashText	buffer: char**	N/A	Choose a random line from the splash.txt file and allocate a new

			buffer to store the line in.
tekCreateMenu	gui: TekGuiComponents*	N/A	Create the menus system. Will call all the sub menu creation functions.
tekDrawMenu	gui: TekGuiComponents*	N/A	Draws the menu. Decides which draw function to call based on the current mode.
tekDeleteMenu	gui: TekGuiComponents*	N/A	Delete the menu system and call the delete functions of the sub menus.
tekRunCleanup	N/A	N/A	Clean up everything that was allocated for the program.
run	N/A	N/A	The actual main function of the program. Sets up the window, the rendering loop, the gui, the physics engine, and everything else.
main	N/A	The exception code, or 0 if there were no exceptions: int	The entrypoint of the code. Mostly just a wrapper around the \ref run function.

TekGUI

Name	Parameters	Return	Description
tekGuiGetOption sColour	option_name: char*, colour_name: char*, colour: vec4	N/A	Get a colour from the options.yml file.
tekGuiLoadWind owDefaults	defaults: TekGuiWindowD efaults*	N/A	Load the default values for a window into a struct. Has default values if nothing specified in options.yml
tekGuiLoadListW indowDefaults	defaults: TekGuiListWindo wDefaults*	N/A	Load the default values from options.yml into a struct. Has default values if nothing specified in options.yml
tekGuiLoadTextB uttonDefaults	defaults: TekGuiTextButto nDefaults*	N/A	Load the default values for a text button from the options.yml file. Has some default values if none

			found.
tekGuiLoadTextInputDefaults	defaults: TekGuiTextInput Defaults*	N/A	Load the text input default values from the options.yml file. Has some default values
tekGuiGetWindowDefaults	defaults: TekGuiWindowDefaults*	N/A	Get the default values for a window gui element.
tekGuiGetListWindowDefaults	defaults: TekGuiListWindowDefaults*	N/A	Get the default values for a list window.
tekGuiGetTextButtonDefaults	defaults: TekGuiTextButtonDefaults*	N/A	Get default values for tek gui button element.
tekGuiGetTextInputDefaults	defaults: TekGuiTextInput Defaults*	N/A	Get default values for text input gui elements.
tekGuiDelete	N/A	N/A	Delete callback for tek gui code.
tekGuiGLLoad	N/A	N/A	Callback for when opengl has loaded for tek gui code.
tekGuilInit	N/A	N/A	Initialisation function for tek gui base / loader code.
tekGuiGetDefaultFont	font: TekBitmapFont**	N/A	Get the default font for the program.

Primitives

Name	Parameters	Return	Description
tekPrimitiveFramebufferCallback	window_width: int, window_height: int	N/A	Framebuffer resize callback for primitives code. Called whenever window is resized.
tekGuiCreateRectangularMesh	point_a: vec2, point_b: vec2, mesh: TekMesh*	N/A	Create a simple rectangular mesh from a top left and bottom right position.
tekGuiCreateImageMesh	width: float, height: float, mesh: TekMesh*	N/A	Create the mesh needed to draw an image - requires there to be texture coordinates as well as positions.
tekDeletePrimitive	N/A	N/A	Delete callback for primitives code,

es			deletes shaders.
tekGLLoadPrimitives	N/A	N/A	The opengl load callback for the primitives code, sets up the shaders needed to draw primitive shapes.
tekInitPrimitives	N/A	N/A	Initialise the primitives code, setting up callbacks.
tekGuiCreateLine	point_a: vec2, point_b: vec2, thickness: float, color: vec4, line: TekGuiLine*	N/A	Create a line that connects two points with a certain thickness.
tekGuiDrawLine	line: TekGuiLine*	N/A	Draw a line to the screen.
tekGuiDeleteLine	line: TekGuiLine*	N/A	Delete a line, freeing any associated memory.
tekGuiCreateOval	point_a: vec2, point_b: vec2, thickness: float, fill: flag, color: vec4, oval: TekGuiOval*	N/A	Create an oval based on two points, the oval will occupy the bounding box specified by the two points.
tekGuiDrawOval	oval: TekGuiOval*	N/A	Draw an oval to the screen.
tekGuiDeleteOval	oval: TekGuiOval*	N/A	Delete an oval, freeing any allocated memory. Archaic function, was used in the noughts and crosses edition.
tekGuiCreateImage	width: float, height: float, texture_filename: char*, image: TekGuilImage*	N/A	Create an image from an image file, and create the mesh needed to draw it.
tekGuiDrawImage	image: TekGuilImage*, x: float, y: float	N/A	Draw an image to the screen at the specified position.
tekGuiDeleteImage	image: TekGuilImage*	N/A	Delete the memory associated with a gui image.

List Window

Name	Parameters	Return	Description
tekGuiListWindowRemoveLookup	window: TekGuiListWindow*, key: char*	N/A	Remove a text mesh from the lookup, freeing any associated memory with that lookup.
tekGuiListCleanup	window: TekGuiListWindow*	N/A	Clean out the lookup table, remove any items that are no longer in the list being displayed.
tekGuiListWindowAddLookup	window: TekGuiListWindow*, text: char*, tek_text: TekText**	N/A	Add a text mesh to the lookup.
tekGuiListWindowGetLookup	window: TekGuiListWindow*, text: char*, tek_text: TekText**	N/A	Get a text mesh by looking up the string it is associated with. If there is no such text, create the text and add it to the lookup.
tekGuiListWindowDeleteLookup	window: TekGuiListWindow*	N/A	Delete the text mesh lookup table, freeing any allocated memory.
tekGuiGetColourBrightness	colour: vec4	The brightness between 0.0 and 1.0: float	Formula to calculate a brightness value for a colour.
tekGuiModifyColourBrightness	original_colour: vec4, final_colour: vec4, delta: float	N/A	Unused method to modify the brightness of a colour. Will adjust the colour in the most suitable direction to have the largest impact on making it look different to before.
tekGuiDrawListWindow	window: TekGuiWindow*	N/A	Draw a list window, drawing each string that is in the list and is visible.
tekGuiSelectListWindow	window: TekGuiWindow*	N/A	Select a list window and bring it to the front of the gui.
tekGuiListWindowGetIndex	window: TekGuiListWindow*, mouse_y: int	The index in the list that is being hovered: int	Get the current index being hovered / clicked based on y coordinate of mouse.

tekGuiListWindowButtonCallback	button: TekGuiButton*, callback_data: TekGuiButtonCallbackData	N/A	Callback for when the active area of a list window is clicked. Will pass on the clicked index to further callbacks.
tekGuiCreateListWindow	window: TekGuiListWindow*, text_list: List*	N/A	Create a list window gui element, which displays a scrollable list of strings. The window will automatically adjust if new strings are added to the list.
tekGuiDeleteListWindow	window: TekGuiListWindow*	N/A	Delete a list window, freeing any memory allocated to the structure.

Box Manager

Name	Parameters	Return	Description
tekGuiBoxDelete	N/A	N/A	Delete callback for box code. Free supporting memory.
tekGuiBoxGLLoad	N/A	N/A	Callback for when opengl loads. Set up the vertex buffer + element buffer + vertex array
tekGuiBoxInit	N/A	N/A	Initialisation function for gui boxes. Set some callbacks and create supporting data structures.
tekGuiCreateBox	box_data: TekGuiBoxData*, index: uint*	N/A	Create a new box to be added to the box buffer.
tekGuiUpdateBox	box_data: TekGuiBoxData*, index: uint	N/A	Update a box in the box buffer with new coordinates. Will overwrite the old mesh data.
tekGuiDrawBox	index: uint, background_colour: vec4, border_colour: vec4	N/A	Draw a generic box to the screen, with a background and border colour. The box to draw is specified by the index in the buffer containing all boxes, this is given by the \ref tekGuiCreateBox method.

Window

Name	Parameters	Return	Description
------	------------	--------	-------------

MAX	a: number, b: number	N/A	Return the largest of two numbers
tekGuiWindowDelete	N/A	N/A	Delete callback for window, freeing any supporting data structures.
tekGuiWindowGLLoad	N/A	N/A	Callback for when opengl loads. Loads window defaults and creates the cursor for when window is moving.
tekGuiWindowInit	N/A	N/A	Initialise some supporting data structures for the window code. Loads some callbacks.
tekGuiGetWindowData	window: TekGuiWindow*, box_data: TekGuiBoxData*	N/A	Get the data for a window box.
tekGuiWindowAddGLMesh	window: TekGuiWindow*, index: uint*	N/A	Create a box mesh for the window and add it to the list of boxes (all stored in one buffer)
tekGuiWindowUpdateGLMesh	window: TekGuiWindow*	N/A	Update the mesh of the background area of the window.
tekGuiBringWindowToFront	window: TekGuiWindow*	N/A	Bring a window to the front so it is rendered above other windows.
tekGuiWindowUpdateButtonHitbox	window: TekGuiWindow*	N/A	Update the position of the title button hitbox according to the window position.
tekGuiWindowCreateTitleText	window: TekGuiWindow*	N/A	Create the visual title text of a window.
tekGuiWindowRecreateTitleText	window: TekGuiWindow*	N/A	Recreate the title text visually of the window.
tekGuiWindowTitleButtonCallback	button_ptr: TekGuiButton*, callback_data: TekGuiButtonCallbackData	N/A	Callback for whenever the title area of a window is clicked. Handles dragging the window.
tekGuiSetWindowTitleBuffer	window: TekGuiWindow*, title: char*	N/A	Update the string buffer that contains the window title, this will allocate or reallocate a buffer as needed, and copy in the title string.

tekGuiSetWindowTitle	window: TekGuiWindow*, title: char*	N/A	Set the title of a window in the gui (not the main window). Title is copied, changing original string does not affect the title.
tekGuiCreateWindow	window: TekGuiWindow*	N/A	Create a new window and add it to the list of all windows.
tekGuiDrawWindow	window: TekGuiWindow*	N/A	Draw a single window, and call any sub draw calls of the window.
tekGuiDrawAllWindows	N/A	N/A	Draw all the windows that have been created.
tekGuiSetWindowPosition	window: TekGuiWindow*, x_pos: int, y_pos: int	N/A	Set the position of a window in pixels.
tekGuiSetWindowSize	window: TekGuiWindow*, width: uint, height: uint	N/A	Set the size of a window in pixels.
tekGuiSetWindowBackgroundColour	window: TekGuiWindow*, colour: vec4	N/A	Set the background colour of a window.
tekGuiSetWindowBorderColour	window: TekGuiWindow*, colour: vec4	N/A	Set the colour of the border of a window.
tekGuiDeleteWindow	window: TekGuiWindow*	N/A	Delete a window, freeing any allocated memory.

Text Button

Name	Parameters	Return	Description
tekGuiGetTextButtonData	button: TekGuiTextButton*, box_data: TekGuiBoxData*	N/A	Get the data needed to draw a box for the button.
tekGuiTextButtonAddGLMesh	text_button: TekGuiTextButton*, index: uint*	N/A	Add box mesh to the list of boxes to be drawn. All box backgrounds are stored in the same vertex buffer So each box usage needs to be registered to find the index of the box in this buffer.

tekGuiTextButtonUpdateGLMesh	text_button: TekGuiTextButton*	N/A	Update the background box mesh of the text button.
tekGuiTextButtonCreateText	button: TekGuiTextButton*	N/A	Create some text to be displayed on the text button.
tekGuiTextButtonRecreateText	button: TekGuiTextButton*	N/A	Recreate the displayed text on a text button.
tekGuiTextButtonCallback	button: TekGuiButton*, callback_data: TekGuiButtonCallbackData	N/A	Callback for text buttons. Passes on the callback to the text button and update hovered status.
tekGuiCreateTextButton	text: char*, button: TekGuiTextButton*	N/A	Create a new text button which is a button that has text.
tekGuiSetTextButtonPosition	button: TekGuiTextButton*, x_pos: uint, y_pos: uint	N/A	Set the pixel position of a text button on the window.
tekGuiSetTextButtonSize	button: TekGuiTextButton*, width: uint, height: uint	N/A	Set the size of a text button in pixels.
tekGuiSetTextButtonText	button: TekGuiTextButton*, text: char*	N/A	Update the text that is displayed on a text button.
tekGuiDrawTextButton	button: TekGuiTextButton*	N/A	Draw a text button to the screen, drawing the background and text. Also shade the button if hovered.
tekGuiDeleteTextButton	button: TekGuiTextButton*	N/A	Delete a text button, freeing any memory allocated.

Button

Name	Parameters	Return	Description
tekGuiGetButtonIndex	button: TekGuiButton*	The index of the button	Get the index of a button in the button list by pointer.

		(possibly 0 if not in the list): uint	
tekGuiCheckButtonOnHitbox	button: TekGuiButton*, check_x: int, check_y: int	1 if the button contains the coordinate, 0 otherwise: int	Check if a position is within a button.
tekGuiButtonMouseButtonCallback	button: int, action: int, mods: int	N/A	Mouse button callback for button code. If click occurs within a button, then send callback to the button and only that button.
tekGuiButtonMouseMovePosCallback	x: double, y: double	N/A	Mouse movement callback for button code. If mouse moves within region of a button, the topmost button will bbe called back.
tekGuiButtonMouseScrollCallback	x_offset: double, y_offset: double	N/A	Mouse scrolling callback for button code. The topmost button will be called back if the scroll occurs in their bounds.
tekGuiButtonDelete	N/A	N/A	Delete callback for button code. Delete supporting data structures.
tekGuiButtonInit	N/A	N/A	Initialisation function for gui button code. Set up callbacks
tekGuiCreateButtonOn	button: TekGuiButton*	N/A	Create a new button. A button is not rendered to the screen, it is just an area that responds to mouse activity.
tekGuiSetButtonPosition	button: TekGuiButton*, x: int, y: int	N/A	Set the position on the window of a button in pixels.
tekGuiSetButtonSize	button: TekGuiButton*, width: uint, height: uint	N/A	Set the size of a button in pixels.
tekGuiBringButtonToFront	button: TekGuiButton*	N/A	Bring a button to the front of the screen, so it is rendered above other buttons

tekGuiDeleteButton	button: TekGuiButton*	N/A	Delete a button, freeing any allocated memory
--------------------	-----------------------	-----	---

Option Window

Name	Parameters	Return	Description
tekGuiLoadOptInt	yml_file: YmlFile*, key: char*, uint_ptr: uint*	N/A	Load a single unsigned integer given a key and a yml file.
tekGuiLoadOptString	yml_file: YmlFile*, key: char*, string: char**	N/A	Load a string from a yml file given a key. NOTE: This function allocates memory which needs to be freed.
tekGuiGetOptionInputType	option_type: char*	The input type, which will be - 1 A.K.A. TEK_UNKNOWWN_INPUT if the string was invalid: flag	Get the type of user input from a string.
tekGuiLoadOptIndex	yml_file: YmlFile*, key: char*, index: double*	N/A	Get the index of an option. The index is a ranking of how far down vertically an option should appear. Kinda like how line numbers work in BASIC.
tekGuiLoadOptByName	yml_file: YmlFile*, key: char*, option: TekGuiOptionsWindowOption*	N/A	Load a single option from a yml file given the name of the input. NOTE: This function will allocate memory that needs to be freed.
tekGuiLoadOptAll	yml_file: YmlFile*, keys: char**, num_keys: uint, options: TekGuiOptionsWindowOption**	N/A	Load all options as described in a yml file.
tekGuiLoadOptAllYml	yml_file: YmlFile*, defaults: TekGuiOptionsWindowDefaults*,	N/A	Load all data for an options window including window metadata from a yml file. NOTE: The options array will be allocated by the function and needs to be

	options: TekGuiOptionsWi ndowOption**, len_options: uint*		freed.
tekGuiReadStrin gOption	window: TekGuiOptionWi ndow*, key: char*, string: char**	N/A	Read a string from a named option. Only provides a reference to the string, don't edit directly!
tekGuiReadNum berOption	window: TekGuiOptionWi ndow*, key: char*, number: double*	N/A	Read a floating point (double) number from a named option.
tekGuiReadBool eanOption	window: TekGuiOptionWi ndow*, key: char*, boolean: flag*	N/A	Read a boolean option from a named option. Will return either 0 or 1.
tekGuiReadVec3 Option	window: TekGuiOptionWi ndow*, key: char*, vector: vec3	N/A	Read a vec3 from a named option. Requires a vec3 to write into.
tekGuiReadVec4 Option	window: TekGuiOptionWi ndow*, key: char*, vector: vec4	N/A	Read a vec4 from a named option. Requires the vec4 to exist to write into.
tekGuiUpdateOpt ionInputText	window: TekGuiOptionWi ndow*, option: TekGuiOption*	N/A	Update the text displayed by an input option based on its type. For example, numbers need to be converted to strings in order to be displayed and such.
tekGuIsOptionIn put	option: TekGuiOption*	0 if not an input, 1 otherwise: flag	Return whether the option is an option input that can be written into.
tekGuiUpdateInp utOption	window: TekGuiOptionWi ndow*, key: char*	N/A	Update a single input option in the array of options

tekGuiWriteStringOption	window: TekGuiOptionWindow*, key: char*, string: char*, len_string: uint	N/A	Write a string into a named option.
tekGuiWriteNumberOption	window: TekGuiOptionWindow*, key: char*, number: double	N/A	Write a number to a named input option.
tekGuiWriteBooleanOption	window: TekGuiOptionWindow*, key: char*, boolean: flag	N/A	Write a boolean value to a named input option.
tekGuiWriteVec3Option	window: TekGuiOptionWindow*, key: char*, vector: vec3	N/A	Write a vec3 to a named input option.
tekGuiWriteVec4Option	window: TekGuiOptionWindow*, key: char*, vector: vec4	N/A	Write a vec4 to a named input option.
tekGuiWriteNumberOptionString	window: TekGuiOptionWindow*, key: char*, number_str: char*	N/A	Write a number option given by a decimal numeric string. Will convert the string to a number, or will be 0 if the string is not a valid number.
tekGuiWriteBooleanOptionString	window: TekGuiOptionWindow*, key: char*, boolean_str: char*	N/A	Write a boolean option given by a string. Will convert the string to a boolean, or will be false if the string is not a valid boolean. Will ignore case and accept either "True", "Yes", or "OK" as being true, anything else is false.
tekGuiWriteVecIndexOptionString	window: TekGuiOptionWindow*, key:	N/A	Write an element of a vector given by a decimal numeric string. Will convert the string to a number, or

	char*, element_str: char*, index: uint		will be 0 if the string is not a valid number. Will write to the index of the vector specified.
tekGuiWriteDefaultValue	window: TekGuiOptionWindow*, name: char*, type: flag	N/A	Writes a default value into an input option. For example, a string -> nullptr, number -> 0.0, boolean -> false.
tekGuiOptionInputCallback	text_input: TekGuiTextInput*, text: char*, len_text: uint	N/A	Callback for when text is inputted into one of the options. The edited text input is specified as the first parameter. Also provides a pointer to the new text and its length. Gives a pointer to the actual text, so you shouldn't overwrite it.
tekGuiButtonInputCallback	button: TekGuiTextButton*, callback_data: TekGuiButtonCallbackData	N/A	Callback for when a button option is clicked. All buttons will link to this callback, but callback receives a pointer to the button which called it. Will simply pass the callback on to the handler specified by the user.
tekGuiCreateLabelOption	label: char*, text_height: uint, option_display: TekGuiOption*, option_index: uint*	N/A	Create a label for the window and add it to the list of displayed items.
tekGuiCreateSingleInput	window: TekGuiOptionWindow*, name: char*, type: flag, input_width: uint, option_display: TekGuiOption*, option_index: uint*	N/A	Create a single input display and add it to the list of displayed items.
tekGuiCreateMultipleInput	window: TekGuiOptionWindow*, name: char*, type: flag, input_width: uint, num_inputs: uint, option_display:	N/A	Create multiple input displays which are all linked to the same name, used for vector inputs

	TekGuiOption*, option_index: uint*		
tekGuiCreateButtonOnOption	window: TekGuiOptionWindow*, name: char*, label: char*, option_display: TekGuiOption*, option_index: uint*	N/A	Create a button type option and add it to the window. Will add to the list of options, and set up callbacks.
tekGuiCreateOptionsOn	window: TekGuiOptionWindow*, name: char*, label: char*, type: flag, text_height: uint, input_width: uint, option_display: TekGuiOption*, option_index: uint*	N/A	Add a chunk of option displays based on a set of parameters. For example, a vec3 option will create 4 option displays, one for the label, and 3 inputs.
tekGuiDrawOptionLabel	option: TekGuiOption*, x_pos: int, y_pos: int	N/A	Draw a label, just a piece of text.
tekGuiDrawOptionInput	option: TekGuiOption*, x_pos: int, y_pos: int	N/A	Draw an option input, which is a text input + link back to the option table.
tekGuiDrawButtonInput	option: TekGuiOption*, x_pos: int, y_pos: int	N/A	Draw a button input, which is just a text button + link back to main window.
tekGuiDrawOption	option: TekGuiOption*, x_pos: int, y_pos: int	N/A	Draw an option display, this call will decide the appropriate method of drawing based on the type.
tekGuiBringOptionToFront	option: TekGuiOption*	N/A	Bring the internal button collider of an option to the front of the button list, so that it will receive mouse

			inputs before other buttons on the screen, and appear to be above them.
tekGuiOptionWindowDrawCallback	window_ptr: TekGuiWindow*	N/A	Draw callback function for option window, called after the base window is drawn.
tekGuiOptionWindowSelectCallback	window_ptr: TekGuiWindow*	N/A	Called every time the base window is brought to the front.
tekGuiCreateBaseWindow	window: TekGuiOptionWindow*, defaults: TekGuiOptionsWindowDefaults*	N/A	Create the base window of the option window. Simple stuff like setting size, position and title.
tekGuiGetOptionDisplaySize	type: flag	The size: uint	Get the size (the number of option displays required) to fit a type of option display.
tekGuiGetOptionsDisplayTotalSize	options: TekGuiOptionsWindowOption*, len_options: uint	The total size: uint	Get the total size of an array of options, for example a label, a vec3 input and a string input.
tekGuiCreateOptionWindow	options_yml: char*, window: TekGuiOptionWindow*	N/A	Create an option window from a yml file.
tekGuiDeleteOption	option: TekGuiOption*	N/A	Free any memory allocated by a displayed option.
tekGuiDeleteOptionWindow	window: TekGuiOptionWindow*	N/A	Free any memory allocated by the option window.

Text Input

Name	Parameters	Return	Description
tekGuiGetTextInputData	text_input: TekGuiTextInput*, box_data: TekGuiBoxData*	N/A	Get the data for drawing a background box from a text input.
tekGuiGetTextInputTextLength	text_input: TekGuiTextInput*	The number of characters being	Get the number of characters currently being displayed to the screen by a text input.

		displayed to the screen: uint	
tekGuiTextInputD ecrementCursor	text_input: TekGuiTextInput*	N/A	Move the cursor backwards, and scroll the text if needed.
tekGuiTextInputI ncrementCursor	text_input: TekGuiTextInput*	N/A	Move the cursor forwards by a single position, scrolling the text if needed.
tekGuiTextInputA ddGLMesh	text_input: TekGuiTextInput*, index: uint*	N/A	Create a new text input box background shape for a new text input.
tekGuiTextInputU pdateGLMesh	text_input: TekGuiTextInput*	N/A	Visually update the text input background area.
tekGuiTextInputG etTextPtr	text_input: TekGuiTextInput*, buffer: char**	N/A	Allocate a buffer of the visible text being displayed.
tekGuiTextInputC reateText	text_input: TekGuiTextInput*	N/A	Create the text mesh to be displayed to the user.
tekGuiTextInputR ecreateText	text_input: TekGuiTextInput*	N/A	Recreate the meshes needed to draw the text so that changes to the contents of the text input are reflected visually.
tekGuiTextInputA dd	text_input: TekGuiTextInput*, codepoint: char	N/A	Add a character to the text input at the cursor position.
tekGuiTextInputR emove	text_input: TekGuiTextInput*	N/A	Remove a character from a text input gui.
tekGuiTextInputC harCallback	codepoint: uint	N/A	Callback for listening for actual typed text, so caps lock gives capital letters and such.
tekGuiFinishTextI nput	text_input: TekGuiTextInput*	N/A	Finish inputting on a text input, reset the text to initial position.
tekGuiTextInputK eyCallback	key: int, scancode: int, action: int, mods: int	N/A	Callback for when a key is pressed to affect text inputs. Listens for enter key, arrow keys etc.
tekGuiTextInputB uttonCallback	button: TekGuiButton*, callback_data:	N/A	Callback for when the text input is clicked.

	TekGuiButtonCall backData		
tekGuiTextInputG LLoad	N/A	N/A	Load text input related things that require opengl. Just loads fonts presently.
tekGuiTextInputI nit	N/A	N/A	Initialise some stuff for text inputs, just callbacks for keys and typing and loading.
tekGuiCreateTextI nput	text_input: TekGuiTextInput*	N/A	Create a new text input gui element.
tekGuiDrawTextI nput	text_input: TekGuiTextInput*	N/A	Draw a text input gui element to the screen.
tekGuiSetTextInp utPosition	text_input: TekGuiTextInput*, x_pos: int, y_pos: int	N/A	Update the position of a text input gui element.
tekGuiSetTextInp utSize	text_input: TekGuiTextInput*, width: uint, height: uint	N/A	Set the width and height of a text input gui element in pixels.
tekGuiSetTextInp utText	text_input: TekGuiTextInput*, text: char*	N/A	Set the text being displayed by a text input.
tekGuiDeleteTextI nput	text_input: TekGuiTextInput*	N/A	Delete a text input gui element, freeing any allocated memory.

Thread Queue

Name	Parameters	Return	Description
threadQueueCre ate	thread_queue: ThreadQueue*, capacity: uint	N/A	Initialise a thread queue. Based on the principle of a lock-free circular queue. NOTE: Single-consumer single-producer
threadQueueDel ete	thread_queue: ThreadQueue*	N/A	Delete a thread queue and free the buffer allocated. NOTE: Only frees the thread queue's used memory, not that of stored pointers.
threadQueueEnq ueue	thread_queue: ThreadQueue*,	0 if the queue is already full,	Enqueue a new item to the thread queue, storing a pointer. NOTE:

	data: void*	1 if the operation was successful: flag	Only to be used by a single producer thread.
threadQueueDequeue	thread_queue: ThreadQueue*, data: void**	0 if the queue is empty, 1 if the operation was successful: flag	Dequeue from a thread queue, returning the stored pointer. NOTE: Only to be used by a single consumer thread.
threadQueuePeek	thread_queue: ThreadQueue*, data: void**	0 if the queue is empty, 1 if the operation was successful: flag	Peek into a thread queue, returning the stored pointer. NOTE: Only to be used by a single consumer thread.
threadQueueIsEmpty	thread_queue: ThreadQueue*	1 if the queue is empty, 0 if not: flag	Determine whether a queue is empty. NOTE: Only to be used by a single consumer thread.

Vector

Name	Parameters	Return	Description
vectorCreate	start_capacity: uint, element_size: uint, vector: Vector*	N/A	Create a vector / resizing array
vectorWriteItem	vector: Vector*, index: uint, item: void*	N/A	Directly write an item to the vector. NOTE: Doesn't perform any safety checks, only used internally to avoid repeating code.
vectorDoubleCapacity	vector: Vector*	N/A	Increase the capacity of the vector by a factor of 2.
vectorAddItem	vector: Vector*, item: void*	N/A	Add an item to the vector. NOTE: The item WILL be copied into the vector, you can safely free anything added to vector afterwards and keep the data here.
vectorSetItem	vector: Vector*, index: uint, item:	N/A	Set an item in the vector at a certain index. NOTE: The item

	void*		WILL be copied into the vector, you can safely free anything added to vector afterwards and keep the data here.
vectorGetItem	vector: Vector*, index: uint, item: void*	N/A	Get an item from the vector. NOTE: The pointer to the item shoud be a buffer large enough to contain the retrieved item. Works best to use the struct you used when setting the element size at initialisation.
vectorGetItemPtr	vector: Vector*, index: uint, item: void**	N/A	Get the pointer to an item in the list. NOTE: Can be better than using vectorGetItem as it avoids making a copy of the data. But if a copy is needed, use the other version.
vectorRemoveItem	vector: Vector*, index: uint, item: void*	N/A	Remove an item from the vector. NOTE: This will shift all the other items in the vector down to fill the gap that is left. If you want to avoid this, just use vectorSetItem with a zeroed struct.
vectorPopItem	vector: Vector*, item: void*	1 if successful, 0 if the vector is empty: flag	Pop an item from the vector, e.g. return last item and remove it from the vector. NOTE: Internally uses vectorGetItem, check for more info.
vectorInsertItem	vector: Vector*, index: uint, item: void*	N/A	Insert an item into a vector at a certain index.
vectorClear	vector: Vector*	N/A	Set all items in the vector to NULL and reset the length to 0. NOTE: This will maintain the previous internal capacity of the vector.
vectorDelete	vector: Vector*	N/A	Delete a vector, freeing the internal list and zeroing the struct.

Queue

Name	Parameters	Return	Description
queueCreate	queue: Queue*	N/A	Initialise a queue struct. NOTE:

			This function requires the queue struct to be allocated already.
queueDelete	queue: Queue*	N/A	Free all memory that a queue has allocated.
queueEnqueue	queue: Queue*, data: void*	N/A	Add a piece of data to the queue.
queueDequeue	queue: Queue*, data: void**	N/A	Retrieve the data stored at the front of the queue, and remove it from the queue.
queuePeek	queue: Queue*, data: void**	N/A	Retrieve the data stored at the front of the queue.
queueIsEmpty	queue: Queue*	0 if the queue is empty, 1 if it is not empty: flag	Check whether a queue is empty.

Bit Set

Name	Parameters	Return	Description
bitsetCreate	num_bits: uint, grows: flag, bitset: BitSet*	N/A	Initialise a BitSet struct by allocating the internal array that will store the bits. NOTE: When using a non-growing bitset, the number of bits available will be rounded to the nearest 64.
bitsetDelete	bitset: BitSet*	N/A	Delete a bitset struct by freeing the internal array, and zeroing the struct.
bitsetGetIndices	bitset_index: uint, array_index: uint*, bit_index: uint*	N/A	Return the two indices that locate a single bit in the bitset.
bitsetDoubleSize	bitset: BitSet*	N/A	Double the capacity of a bitset, used when space has run out.
bitsetSetValue	bitset: BitSet*, index: uint, value: char	N/A	Helper method to update the value of a bit at a certain index, and grow array if needed.
bitsetSet	bitset: BitSet*, index: uint	N/A	Set a bit to 1 at the specified index.
bitsetUnset	bitset: BitSet*,	N/A	Set a bit to 0 at the specified

	index: uint		index.
bitsetGet	bitset: BitSet*, index: uint, value: flag*	N/A	Get the value of the bit at a specific index.
bitsetGet1DIndex	x: uint, y: uint	The 1D index: uint	Helper function to get a 1D index from a 2D coordinate. The index snakes across a grid, so if the bitset expands, the coordinates still map to the same index in the array. All credit goes to me at 2:30am on a random saturday for this idea.
bitsetSet2D	bitset: BitSet*, x: uint, y: uint	N/A	Set the bitset at a coordinate (x, y)
bitsetUnset2D	bitset: BitSet*, x: uint, y: uint	N/A	Unset the bitset at a coordinate (x, y)
bitsetGet2D	bitset: BitSet*, x: uint, y: uint, value: flag*	N/A	Get the bitset at a coordinate (x, y)
bitsetClear	bitset: BitSet*	N/A	Set all bits to zero.

YAML

Name	Parameters	Return	Description
ymlWordToString	word: Word*, string: char**	N/A	Allocate memory to copy the content of a yml word into. NOTE: This function allocates memory which needs to be freed.
YML_CREATE_ FUNC	param_name: ?, yml_data: ?	N/A	Allocate a new YmlData struct given input data. NOTE: This function allocates memory which needs to be freed.
YML_CREATE_ TOKEN_FUNC	token: ?, yml_data: ?	N/A	Allocate a new YmlData struct using a token. NOTE: This function allocates memory which needs to be freed.
ymlCreateAutoD ataToken	token: Token*, yml_data: YmlData**	N/A	A function to automatically allocate a YmlData struct based on the type of the token. This function allocates memory which needs to

			be freed.
ymlDataToString	yml_data: YmlData*, string: char**	N/A	Copy the string data of a YmlData struct into a newly allocated buffer. NOTE: This function allocates memory which needs to be freed.
ymlDataToInteger	yml_data: YmlData*, integer: long*	N/A	Copy the integer data of a YmlData struct into an existing integer.
ymlDataToFloat	yml_data: YmlData*, number: double*	N/A	Copy the double data of a YmlData struct into an existing double.
YML_GET_LIST_FUNC	yml_list: ?, index: ?, param_name: ?	N/A	Copy the data of a YmlData list at a certain index. NOTE: This function may allocate memory which needs to be freed. (if using strings)
YML_ARRAY_LIST_OOP_FUNC	yml_list: ?, array: ?	N/A	A helper function to fill an existing array by converting a List of YmlData strings. This function requires an array with the same size as the list to already be allocated.
YML_ARRAY_FILL_FUNC	yml_list: ?, array: ?, len_array: ?	N/A	Copy the data of a YmlData list into a newly allocated array. NOTE: This function allocated memory which needs to be freed.
ymlCreateListData	list: List*, yml_data: YmlData**	N/A	Allocate a YmlData struct given a list. NOTE: This function allocates memory which needs to be freed. The list is only stored as a reference, and not copied. The list should therefore not be deleted once stored in the YmlData.
ymlDeleteData	yml_data: YmlData*	N/A	A function to free a YmlData struct based on its type. NOTE: This function attempts to free allocated memory: only YmlData structs created by the other functions should be freed with this function.
ymlDelete	yml: YmlFile*	N/A	Free a previously allocated YML file. NOTE: This function should

			only be called after ymlCreate() or similar, as it will attempt to free memory.
ymlCreate	yml: YmlFile*	N/A	Instantiate a YmlFile struct.
VA_ARGS_TO_LIST	list_name: ?, va_name: ?, va_type: ?, final_param: ?	N/A	Macro template for reading through variadic arguments until reaching nullptr, and adding each va_arg to a list.
ymlGetList	yml: YmlFile*, data: YmlData**, keys: List*	N/A	Takes a list of keys in order and returns the data stored under that key. If we had a YAML file that looked something like this: cpu: clock_speed: 3.8GHz In order to access the CPU's clock speed, we would pass in a list of: @code keys = {"cpu", "clock_speed"} NOTE: This function does NOT copy the data, once yml is freed, the YmlData is also gone.
ymlGetVA	yml: YmlFile*, data: YmlData**, ...: ?	N/A	Get a piece of yml data by variadic arguments.
ymlGetKeysList	yml: YmlFile*, yml_keys: char***, num_keys: uint*, keys: List*	N/A	Takes a list of keys in order and returns the names of all keys under that name. If we had a YAML file that looked something like this: computer: cpu: manufacturer: intel clock_speed: 3.8GHz overclocked: false. In order to access keys under cpu, we would use keys = {"computer", "cpu"} This would return an array of {"manufacturer", "clock_speed", "overclocked"} NOTE: This function will allocate an array, overwriting the pointer specified. This array needs to be freed.
ymlGetKeysVA	yml: YmlFile*, yml_keys: char***, num_keys: uint*, ...: ?	N/A	Get a list of sub keys from a yml data structure by variadic arguments.

ymlSetList	yml: YmlFile*, data: YmlData*, keys: List*	N/A	Takes a list of keys in order and sets the data stored under that key. If we had a YAML file that looked something like this: cpu: clock_speed: 3.8GHz. In order to set the CPU's clock speed, we would pass in a list of: keys = {"cpu", "clock_speed"}
ymlSetVA	yml: YmlFile*, data: YmlData*, ...: ?	N/A	Set an item in a YML data structure by variadic arguments.
ymlRemoveList	yml: YmlFile*, keys: List*	N/A	Takes a list of keys in order and deletes the data stored under that key. If we had a YAML file that looked something like this: cpu: clock_speed: 3.8GHz. In order to delete the CPU's clock speed, we would pass in a list of: keys = {"cpu", "clock_speed"}
ymlRemoveVA	yml: YmlFile*, ...: ?	N/A	Remove item from YML using variadic arguments to specify the item to remove.
isWhitespace	c: char	1 if whitespace, 0 if not: int	Determine if a character is whitespace.
ymlSplitText	buffer: char*, buffer_size: uint, split_text: List*	N/A	Split a buffer of text into individual words
ymlThrowSyntax	word: Word*	N/A	Throw a yml exception, and create a custom message based on the word and line number.
ymlDetectType	word: Word*	Word type: flag	Detect the word type
TOKEN_CREAT E_FUNC	word: ?, token: ?	N/A	Create a token given a word. NOTE: This function allocates memory that needs to be freed.
ymlCreateIndent Token	type: flag, token: Token**	N/A	Create an indent token given a type. NOTE: This function allocates memory that needs to be freed.

ymlUpdateIndent	indent: uint*, word: Word*, tokens: List*, indent_stack: Stack*	N/A	Update the current indent we are at while parsing a yml file. This works not by tracking the number of spaces, but the number of times that the number of spaces increased since the start of the file. For a file such as depth1: depth2: depth3: depth4: "something" The depth stack would look something like {0, 2, 5, 9} Representing the number of spaces at each indent. The size of the stack - 1 (which is 3 here) represents the current indent of the file. If we updated the file to look like depth1: depth2: depth3: depth4: "something" depth5: "bad formatting" The indent stack could be popped repeatedly to find that the current number of spaces (7) never occurred before, and so this formatting is invalid.
ymlCreateTokensStack	split_text: List*, tokens: List*, indent_stack: Stack*	N/A	A function responsible for parsing a Word List, and converting this into a series of tokens that can be processed into a yml data structure.
ymlCreateTokens	split_text: List*, tokens: List*	N/A	See ymlCreateTokensStack
ymlFromTokensListAddList	prev_item: ListItem**, item: ListItem**, yml_list: List*, yml_data: YmlData**	N/A	Helper function to turn a list of tokens into a single YmlData struct. NOTE: This function allocates memory that needs to be freed.
ymlFromTokensList	tokens: List*, yml: YmlFile*, keys_list: List*	N/A	A function to read a series of tokens into YmlData structs.
ymlFromTokens	tokens: List*, yml: YmlFile*	N/A	See ymlFromTokensList
ymlPrintData	data: YmlData*	N/A	Format and print YmlData NOTE: This function will print with a

			newline character at the end.
ymlPrintIndent	yml: YmlFile*, indent: uint	N/A	A helper function to be called recursively, tracking the current indentation that we are printing at.
ymlPrint	yml: YmlFile*	N/A	Formats and prints yml file.
ymlReadFile	filename: char*, yml: YmlFile*	N/A	Take a filename and read its content into a yml data structure.

Stack

Name	Parameters	Return	Description
stackCreate	stack: Stack*	N/A	Create a stack from an existing struct. NOTE: Not strictly necessary, essentially just zeroes out the struct.
stackDelete	stack: Stack*	N/A	Delete a stack, by freeing all nodes of the stack. NOTE: Doesn't free any of the stored data, just the structure itself. Freeing data should be done manually.
stackPush	stack: Stack*, data: void*	N/A	Push an item onto the stack. NOTE: This will not copy the data that the pointer points to, it will only store the pointer. So beware of the pointer being freed while stack still exists.
stackPop	stack: Stack*, data: void**	N/A	Pop an item from the stack.
stackPeek	stack: Stack*, data: void**	N/A	Peek the stack / return item on the top of the stack without removing it.

Hash Table

Name	Parameters	Return	Description
hashtableDelete	hashtable: HashTable*	N/A	Delete a hashtable, freeing all the memory that the hashtable functions allocated. NOTE: This does not free any user allocated memory, for example pointers that are stored in the hashtable.

hashtableCreate	hashtable: HashTable*, length: uint	N/A	Create a hashtable. The length is relatively unimportant, setting it too small just means more rehashes.
hashtableHash	hashtable: HashTable*, key: char*, hash: uint*	N/A	Create a hash index for a hashtable given a string. Essentially works by summing ASCII values of each character, modulo length of internal array.
hashtableCreateNode	hashtable: HashTable*, key: char*, out_ptr: HashNode**, hash: uint	N/A	Create a node for a hashtable, and insert it into the hashtable correctly.
hashtableGetNode	hashtable: HashTable*, key: char*, node_ptr: HashNode**	N/A	Check if a node exists in the hashtable, and if so return a pointer to this node. NOTE: The retrieved node is not a copy, and will be freed once the hashtable is freed.
hashtableGetKeys	hashtable: HashTable*, keys: char***	N/A	Return a list of keys that make up the hashtable. NOTE: The function will allocate an array at the pointer specified that needs to be freed. The length of this array will be hashtable.num_items. (hashtable.length specifies the length of the internal array).
hashtableGetValues	hashtable: HashTable*, values: void***	N/A	Return a list of values stored in the hashtable. NOTE: The function will allocate memory to store the values in, which needs to be freed. The order of the values will match the order of keys retrieved by hashtableGetKeys.
hashtableRehash	hashtable: HashTable*, new_length: uint	N/A	Resize the hashtable to be a new size, copying the items into new locations. NOTE: By the nature of how hashtables work, the order of items from hashtableGetKeys/Values will change after rehashing.
hashtableTooFull	hashtable:	1 if the internal	Returns whether a hashtable is

	HashTable*	array is more than 75% full, 0 otherwise: flag	more than 75% full, at which point the number of collisions makes the hashtable slower, requiring the linked lists to be traversed more often.
hashtableGet	hashtable: HashTable*, key: char*, data: void**	N/A	Get an item from a hashtable by key.
hashtableSet	hashtable: HashTable*, key: char*, data: void*	N/A	Set the value of the hashtable at a certain key.
hashtableRemove	hashtable: HashTable*, key: char*	N/A	Remove the data stored at a certain key in the hashtable. NOTE: This does not free the pointer stored at this key, data stored here should be freed first.
hashtableHasKey	hashtable: HashTable*, key: char*	1 if the item exists, 0 if it does not or an error occurred while looking for it: flag	Check whether an item exists in a hashtable.
hashtablePrint	hashtable: HashTable*	N/A	Print the data about a hashtable, which will print something like HashNode* internal = 0x123456789 unsigned int length = 100
hashtablePrintItems	hashtable: HashTable*	N/A	Print a hashtable in a nice format, something like { "key_1": 0x12987123 "key_2": 0x12312344 "key_3": 0x11245453 }
hashtablePrintInternal	hashtable: HashTable*	N/A	Print out the internal array of the hashtable, which will be a mix of 0s and pointers to linked lists.

File

Name	Parameters	Return	Description
getFileSize	filename: char*, file_size: uint*	N/A	Return the size of a file in bytes.

readFile	filename: char*, buffer_size: uint, buffer: char*	N/A	Read a file into an existing buffer. NOTE: Buffer is required to be large enough for the file. Use the 'getFileSize(...)' function to find the size, and then allocate memory.
writeFile	buffer: char*, filename: char*	N/A	Write a string buffer into a file. This will create a file if one doesn't exist, and will overwrite if it does.
addPathToFile	directory: char*, filename: char*, result: char**	N/A	Combine a directory and a file name into a single string. For example, directory="/usr/tekphys/" filename="file.txt" -> result="/usr/tekphys/file.txt"
fileExists	filename: char*	0 if it does not exist, 1 if it does: flag	Check if a file exists.

Exception

Name	Parameters	Return	Description
tekAddException	exception_code: int, exception_name: char*	N/A	Add a new exception to the list of named exceptions.
tekInitExceptions	N/A	N/A	Initialise exceptions, adding human readable names to the list of exception names.
tekCloseExceptions	N/A	N/A	Frees all the names of the exceptions that were created.
tekPrintException	N/A	N/A	Print out the last exception that occurred, including stack trace.
tekSetException	exception_code: int, exception_line: int, exception_function: char*, exception_file: char*, exception_message: char*	N/A	Record that an exception has occurred, based on line number, function and file. NOTE: Should be used mostly with the 'tekThrow(...)' macro.

tekTraceException	exception_code: int, exception_line: int, exception_function: char*, exception_file: char*	N/A	Function used to add to the stack trace of an exception. NOTE: Should be used only with the 'tekChainThrow(...)' macro.
-------------------	---	-----	---

Priority Queue

Name	Parameters	Return	Description
priorityQueueCreate	queue: PriorityQueue*	N/A	Initialise a priority queue struct. NOTE: This function requires the priority queue struct to be allocated already.
priorityQueueDelete	queue: PriorityQueue*	N/A	Free all memory that a priority queue has allocated.
priorityQueueEnqueue	queue: PriorityQueue*, priority: double, data: void*	N/A	Enqueue a piece of data into the priority queue. Data with the lowest priority value will be dequeued first.
priorityQueueDequeue	queue: PriorityQueue*, data: void**	0 if the queue is empty, 1 otherwise: flag	Dequeue a piece of data from the front of the priority queue.
priorityQueuePeek	queue: PriorityQueue*, data: void**	0 if the queue is empty, 1 otherwise: flag	Peek an item at the front of the priority queue.
priorityQueueIsEmpty	queue: PriorityQueue*	0 if the queue is empty, 1 if it is not empty: flag	Check whether a priority queue is empty.

Test Suite

Name	Parameters	Return	Description
tekTestCreate	test_name: ?	N/A	Create a creation function for a suite. Called every test.
tekTestDelete	test_name: ?	N/A	Create a cleanup function for a test suite.
tekTestFunc	test_name: ?,	N/A	Create a new function in a test

	test_part: ?		suite.
tekRunSuite	test_name: ?, test_part: ?, test_context: ?	N/A	Run a suite of tests
tekAssert	expected: ?, actual: ?	N/A	Assert if two things are equal, and print if they are or are not.
tekSilentAssert	expected: ?, actual: ?	N/A	Assert if two things are equivalent, but only alert if they are not equal.

List

Name	Parameters	Return	Description
listCreate	list: List*	N/A	Create a list using an existing but empty List struct. NOTE: Not strictly needed, but recommended incase struct was allocated using malloc() and contains bogus data.
listDelete	list: List*	N/A	Delete a list by freeing all allocated nodes. NOTE: Does not free any of the data added to the list, only the list structure itself. Anything stored in the list should be manually freed, or with \ref listFreeAllData'
listFreeAllData	list: List*	N/A	Call 'free(...)' on all data in the list.
listAddItem	list: List*, data: void*	N/A	Add an item to the list. NOTE: Will only copy the pointer into the list, but not actually the data. So don't try and add stack variables to it and then pass on to another function, etc.
listSetItem	list: List*, index: uint, data: void*	N/A	Update the data stored at an index of a list. NOTE: Data is not copied, only stored as a pointer.
listInsertItem	list: List*, index: uint, data: void*	N/A	Insert an item at a specific index into the list. NOTE: Will only copy the pointer into the list, but not actually the data. So don't try and add stack variables to it and then pass on to another function, etc.
listGetItem	list: List*, index:	N/A	Get an item from the list.

	uint, data: void**		
listPopItem	list: List*, data: void**	N/A	Pop an item from the end of the list.
listRemoveItem	list: List*, index: uint, data: void**	N/A	Remove an item from the list.
listMoveItem	list: List*, old_index: uint, new_index: uint	N/A	Move an item at one index so that it is now stored at a different index.
listPrint	list: List*	N/A	Print out the data pointer of each item in the list in hexadecimal format. NOTE: Not very useful unless debugging. For more useful output, your own function is needed as the list has no idea what datatype you are storing.

Text

Name	Parameters	Return	Description
tekTextFramebuff erCallback	width: int, height: int	N/A	Framebuffer callback for text code. Update a projection matrix for use in text shader.
tekGLLoadTextE ngine	N/A	N/A	Callback for when opengl is loaded. Creates the shader that allows text to be drawn and which is shared between different text being drawn.
tekDeleteTextEn gine	N/A	N/A	Delete callback for text engine, delete allocated memory.
tekInitTextEngine	N/A	N/A	Initialisation function for loading text engine.
tekGenerateText MeshData	text: char*, len_text: uint, size: uint, font: TekBitmapFont*, tek_text: TekText*, vertices: float**, len_vertices_ptr: uint*, indices: uint**,	N/A	Generate a mesh that represents some lettering. Sorts out where the vertices should go, new line spacing, texture atlas coordinates etc.

	<code>len_indices_ptr: uint*</code>		
<code>tekCreateText</code>	<code>text: char*, size: uint, font: TekBitmapFont*, tek_text: TekText*</code>	N/A	Create a new TekText structure using a font, some text and a size.
<code>tekUpdateText</code>	<code>tek_text: TekText*, text: char*, size: uint</code>	N/A	Update a TekText struct to retain the same font, but redraw with different text and/or a different size.
<code>tekDrawColouredText</code>	<code>tek_text: TekText*, x: float, y: float, colour: vec4</code>	N/A	Draw text at a point with a specific colour.
<code>tekDrawColouredRotatedText</code>	<code>tek_text: TekText*, x: float, y: float, colour: vec4, rot_x: float, rot_y: float, angle: float</code>	N/A	Draw text of a specific colour at a coordinate, and rotate by an angle around a different point.
<code>tekDrawText</code>	<code>tek_text: TekText*, x: float, y: float</code>	N/A	Draw some text to the screen at a position.
<code>tekDeleteText</code>	<code>tek_text: TekText*</code>	N/A	Delete a text struct, freeing any allocated memory.

Mesh

Name	Parameters	Return	Description
<code>tekCreateBuffer</code>	<code>buffer_type: GLenum, buffer_data: void*, buffer_size: long, buffer_usage: GLenum, buffer_id: uint*</code>	N/A	Create a buffer in opengl and fill it with data.
<code>tekGenerateVertexAttributes</code>	<code>layout: int*, len_layout: uint</code>	N/A	Generate the vertex attributes for a mesh based on the layout array.
<code>tekCreateMesh</code>	<code>vertices: float*, len_vertices: uint</code>	N/A	Create a mesh from a collection of vertices.

	<code>len_vertices: long, indices: uint*, len_indices: long, layout: int*, len_layout: uint, mesh_ptr: TekMesh*</code>		arrays. Vertices is the array of actual vertex data. Indices is an array that says the order to access the vertex array in to draw the mesh. The layout describes the length of each chunk of data in the vertex array, e.g. a position vec3, a normal vec3 and a vec2 texture coordinate would have a layout of {3, 3, 2}.
<code>STRING_CONV_FUNC</code>	<code>func_name: ?, func_type: ?, conv_func: ?, conv_type: ?</code>	N/A	
<code>ARRAY_BUILD_FUNC</code>	<code>func_name: ?, array_type: ?, conv_func: ?</code>	N/A	Build an array from a backwards list.
<code>tekCreateMeshArrays</code>	<code>vertices: List*, indices: List*, layout: List*, vertex_array: float*, index_array: uint*, layout_array: int*</code>	N/A	Convert the lists produced by <code>tekReadMeshLists</code> into arrays. Very useful because the lists are produced in reverse order and need to be flipped. And the lists are linked lists which cannot be used as arrays. FAQ: Where is the length? Its stored in the lists already... Does this allocate the lists for me? No, because then its easier to free everything if this fails.
<code>tekReadMeshLists</code>	<code>buffer: char*, vertices: List*, indices: List*, layout: List*, position_layout_index: uint*</code>	N/A	Read a mesh file into lists. The file should be loaded into a buffer and the lists should already be created and ready to use. Should only really be used in \ref <code>tekReadMeshArrays</code> . IMPORTANT: for linked list speed, the items are inserted at the start not the end. So the items come out back to front. Haha. So please dont use this function anywhere else.
<code>tekReadMeshArrays</code>	<code>filename: char*, vertex_array:</code>	N/A	Read a mesh file into arrays.

	<code>float**, len_vertex_array: uint*, index_array: uint**, len_index_array: uint*, layout_array: int**, len_layout_array: uint*, position_layout_i ndex: uint*</code>		
tekReadMesh	<code>filename: char*, mesh_ptr: TekMesh*</code>	N/A	Read a file and write data into a mesh.
tekDrawMesh	<code>mesh_ptr: TekMesh*</code>	N/A	Draw a mesh to the screen. Requires setting a shader/material first in order to render anything.
tekRecreateMes h	<code>mesh_ptr: TekMesh*, vertices: float*, len_vertices: long, indices: uint*, len_indices: long, layout: int*, len_layout: uint</code>	N/A	Recreate an existing mesh and provide a new set of vertices, indices and layout. Can set any parameter to be NULL if you want to retain the old data for that.
tekDeleteMesh	<code>mesh_ptr: TekMesh*</code>	N/A	Delete a mesh by deleting any opengl buffers that were allocated for it.

Manager

Name	Parameters	Return	Description
tekAddFramebuff erCallback	<code>callback: TekFramebufferC allback</code>	N/A	Add a framebuffer callback which will be called whenever the framebuffer/window is resized.
tekManagerFram ebufferCallback	<code>window: GLFWwindow*, width: int, height: int</code>	N/A	Add a framebuffer callback which is called every time the window is resized. This gives the new size of the portion of the window which is being rendered, and doesn't

			include borders.
tekAddDeleteFunc	delete_func: TekDeleteFunc	N/A	Add a delete function which is called when the program is terminating.
tekAddGLLoadFunc	gl_load_func: TekGLLoadFunc	N/A	Add a GL load function, this is called after glfw + glad have been initialised.
tekManagerKeyCallback	window: GLFWwindow*, key: int, scancode: int, action: int, mods: int	N/A	The main callback for all key press events, goes on to call any user-created key callbacks.
tekManagerCharCallback	window: GLFWwindow*, codepoint: uint	N/A	The main callback for char events, e.g. listen for typing with shift/caps lock affecting keys. Goes on to call all user created char callbacks.
tekManagerMouseMoveCallback	window: GLFWwindow*, x: double, y: double	N/A	The main callback for mouse movement events that will go on to trigger all user-created callbacks of this type.
tekManagerMouseButtonCallback	window: GLFWwindow*, button: int, action: int, mods: int	N/A	The main callback for mouse buttons that will go on to trigger user-created callbacks.
tekManagerMouseScrollCallback	window: GLFWwindow*, x_offset: double, y_offset: double	N/A	The root callback for mouse scroll events, which triggers all user-created events.
tekAddKeyCallback	callback: TekKeyCallback	N/A	Add a callback for every time a key is pressed on the keyboard.
tekAddCharCallback	callback: TekCharCallback	N/A	Add a 'char' callback, which is called whenever the user types. It is affected by shift/caps lock etc.
tekAddMousePosCallback	callback: TekMousePosCallback	N/A	Add a callback that is triggered every time the mouse is moved.
tekAddMouseButtonCallback	callback: TekMouseButton	N/A	Add a mouse button callback that is triggered every time the mouse

	Callback		is clicked.
tekAddMouseScrolCallback	callback: TekMouseScroll Callback	N/A	Add a callback that will be triggered every time the mouse is scrolled.
tekSetCursor	cursor_mode: flag	N/A	Set the mode of the cursor. Should be one of two modes: DEFAULT_CURSOR or CROSSHAIR_CURSOR, assumes default cursor if invalid cursor mode specified.
tekSetWindowColour	cocolour: vec3	N/A	Set the colour of the window background when nothing is drawn.
tekSetDrawMode	draw_mode: flag	N/A	Set the draw mode of the window, should be one of: DRAW_MODE_NORMAL - draw everything as expected. DRAW_MODE_GUI - draw everything as gui. This changes how depth is rendered, gui elements are all rendered at the same level. Typical depth test will not work, so setting gui mode will force new things to appear above old things.
tekInit	window_name: char*, window_width: int, window_height: int	N/A	Initialise the TekPhysics window.
tekRunning	N/A	1 if the window is still open, 0 if the X button has been pressed: flag	Return whether the window should remain open.
tekUpdate	N/A	N/A	Swap buffers and poll events. Should be called every frame / every loop of the main loop.
tekDelete	N/A	N/A	Free memory allocated by supporting structures for the manager code, for example

			different callback function lists.
tekGetWindowSize	window_width: int*, window_height: int*	N/A	Get the current size of the window.
tekSetMouseMode	mouse_mode: flag	N/A	Set the mouse mode of the window. Can be one of two types: MOUSE_MODE_CAMERA - make the mouse disappear and unable to leave the screen. MOUSE_MODE_NORMAL - make the mouse behave as expected. If neither one is used, then normal mode is assumed.

Entity

Name	Parameters	Return	Description
tekEntityDelete	N/A	N/A	The cleanup function for the entity code. Deletes caches used for materials and meshes.
tekEntityInit	N/A	N/A	Initialise some surrounding helper structures for entities, notably caches for meshes and materials.
REQUEST_FUNC	func_name: ?, func_type: ?, param_name: ?, create_func: ?, delete_func: ?	N/A	A template function for creating/caching things loaded from files. Used to make a cache of materials and meshes.
tekCreateEntity	mesh_filename: char*, material_filename: char*, position: vec3, rotation: vec4, scale: vec3, entity: TekEntity*	N/A	Create a new entity from a mesh and material file, along with other data. mesh file type = .tmsh, material file type = .tmat
tekUpdateEntity	entity: TekEntity*, position: vec3, rotation: vec4	N/A	Update an entity with a new position and rotation.
tekDrawEntity	entity: TekEntity*, camera:	N/A	Draw an entity to the screen from the perspective of a camera. Also

	TekCamera*		use the material associated with the entity.
tekNotifyEntityMaterialChange	N/A	N/A	Notify the materials renderer that the material has been changed without using the material code directly.

Font

Name	Parameters	Return	Description
tekCreateFreeType	N/A	N/A	Initialise the freetype library and allow fonts to be loaded and used.
tekDeleteFreeType	N/A	N/A	Delete the freetype library once font loading is finished.
tekCreateFontFace	filename: char*, face_index: uint, face_size: uint, face: FT_Face*	N/A	Create a font face from a filename and a face index. Each font file has different faces e.g. bold, italic. The face index specifies which one of these faces you want.
tekGetGlyphSize	face: FT_Face*, glyph: uint, glyph_width: uint*, glyph_height: uint*	N/A	Get the size of a glyph without loading the bitmap data of the glyph.
tekTempLoadGlyph	face: FT_Face*, glyph_id: uint, glyph: TekGlyph*, glyph_data: byte**	N/A	Load a glyph temporarily into memory, get the size and bitmap data of the glyph.
tekGetAtlasSize	face: FT_Face*, atlas_size: uint*	N/A	Get the size of the atlas required to contain a font face.
tekCreateFontAtlasData	face: FT_Face*, atlas_size: uint, atlas_data: byte**, glyphs: TekGlyph*	N/A	Create the font atlas and update the glyph data for each character once it has been added to the font atlas.
tekCreateFontAtlasTexture	face: FT_Face*, texture_id: uint*, atlas_size: uint*,	N/A	Create the font atlas texture, this is a texture that contains every character in a tightly packed area.

	glyphs: TekGlyph*		Also returns an array of glyphs, which store where on the atlas each character is.
tekCreateBitmapFont	filename: char*, face_index: uint, face_size: uint, bitmap_font: TekBitmapFont*	N/A	Create a new bitmap font from a true type font file. The files contain multiple faces such as italic, bold. So you need to also specify which font you want to use with a font index.
tekDeleteFontFace	N/A	N/A	Delete a font face, wrapper around normal method to keep naming convention.

Texture

Name	Parameters	Return	Description
tekCreateTexture	filename: char*, texture_id: uint*	N/A	Create a new texture from an image file, and return the id of the newly created texture. It is recommended to have square textures with side lengths that are powers of 2. e.g. 128x128, 512x512.
tekBindTexture	texture_id: uint, texture_slot: byte	N/A	Bind a texture to a specified texture slot, allows the texture to be accessed in shaders.
tekDeleteTexture	texture_id: uint	N/A	Delete a texture using its id using opengl.

Material

Name	Parameters	Return	Description
tekCreateVecUniform	hashtable: HashTable*, num_items: uint, keys_order: char**, uniform: TekMaterialUniform*	N/A	Create a uniform for vector data that is made of multiple yml key value pairs.
tekCreateUniform	uniform_name: char*, data_type: flag, data: void*, uniform:	N/A	Create a single uniform to add to the uniform array in the material.

	TekMaterialUnifo rm**		
tekDeleteUniform	uniform: TekMaterialUnifo rm*	N/A	Delete a material uniform from memory.
tekCreateMateria l	filename: char*, material: TekMaterial*	N/A	Create a material from a file, a collection of shaders and uniforms to texture a mesh.
tekBindMaterial	material: TekMaterial*	N/A	Bind (use) a material, so that all subsequent draw calls will be drawn with this material.
tekMaterialHasU niformType	material: TekMaterial*, uniform_type: flag	1 if the material has that type, 0 otherwise: flag	Return whether a material has a specific type of uniform. Useful to check if a material has a projection matrix etc.
MATERIAL_BIN D_UNIFORM_F UNC	N/A	N/A	Template function for binding uniforms.
tekDeleteMateria l	material: TekMaterial*	N/A	Delete a material, freeing any allocated memory.

Camera

Name	Parameters	Return	Description
tekUpdateCamer aProjection	camera: TekCamera*	N/A	Update the projection matrix of the camera.
tekUpdateCamer aView	camera: TekCamera*	N/A	Update the view matrix of the camera.
tekCameraFram ebufferCallback	framebuffer_widt h: int, framebuffer_heig ht: int	N/A	Called when the framebuffer size changes, e.g. when the user resizes the window.
tekCameraDelet eFunc	N/A	N/A	Called when program exits, cleans up some structures.
tekCameralinit	N/A	N/A	Initialise some structures needed to have cameras.
tekSetCameraPo sition	camera: TekCamera*, position: vec3	N/A	Update the position of a camera and update any matrices this would affect.

tekSetCameraRotation	camera: TekCamera*, rotation: vec3	N/A	Set the rotation of the camera and update the matrices it affects.
tekCreateCamera	camera: TekCamera*, position: vec3, rotation: vec3, fov: float, near: float, far: float	N/A	Create a camera struct, a container for some information about a camera + matrices needed to render with this camera.

Shader

Name	Parameters	Return	Description
tekDeleteShader	shader_id: uint	N/A	Delete a shader using its id.
tekCreateShader	shader_type: GLenum, shader_filename: char*, shader_id: uint*	N/A	Compile a piece of shader code contained within a file and return an id of this shader.
tekBindShaderProgram	shader_program_id: uint	N/A	Bind (use) a shader program using its id. All subsequent draw calls will then use this shader.
tekDeleteShaderProgram	shader_program_id: uint	N/A	Delete a shader program using its id.
tekCreateShaderProgramViFi	vertex_shader_id : uint, fragment_shader_id: uint, shader_program_id: uint*	N/A	Create a shader program from a vertex shader and fragment shader id.
tekCreateShaderProgramViGiFi	vertex_shader_id : uint, geometry_shader_id: uint, fragment_shader_id: uint, shader_program_id: uint*	N/A	Create a shader program from a vertex shader, geometry shader and fragment shader id.
tekCreateShaderProgramVF	vertex_shader_filename: char*, fragment_shader_filename: char*,	N/A	Create a shader program from a vertex and fragment shader.

	shader_program_id: uint*		
tekCreateShaderProgramVGF	vertex_shader_filename: char*, geometry_shader_filename: char*, fragment_shader_filename: char*, shader_program_id: uint*	N/A	Create a shader program from a vertex, geometry and fragment shader.
tekGetShaderUniformLocation	shader_program_id: uint, uniform_name: char*, uniform_location: int*	N/A	Get the location of a shader uniform by name.
tekShaderUniformInt	shader_program_id: uint, uniform_name: char*, uniform_value: int	N/A	Write an integer into a shader uniform.
tekShaderUniformFloat	shader_program_id: uint, uniform_name: char*, uniform_value: float	N/A	Write a float into a shader uniform.
tekShaderUniformVec2	shader_program_id: uint, uniform_name: char*, uniform_value: vec2	N/A	Write a 2 vector into a shader uniform.
tekShaderUniformVec3	shader_program_id: uint, uniform_name: char*, uniform_value: vec3	N/A	Write a 3 vector to a shader uniform.
tekShaderUniformMat4	shader_program_id: uint	N/A	Write a shader uniform for a 4

mVec4	<code>_id: uint, uniform_name: char*, uniform_value: vec4</code>		vector.
tekShaderUniformMat4	<code>shader_program_id: uint, uniform_name: char*, uniform_value: mat4</code>	N/A	Update a shader uniform of mat4 type.

Engine

Name	Parameters	Return	Description
RECV_FUNC	N/A	N/A	Template for generating receive functions for thread queues.
PUSH_FUNC	N/A	N/A	Template function for generating thread queue push functions.
CAM_UPDATE_FUNC	N/A	N/A	Template for generating functions for updating the camera. Actually obsolete now but too late to remove.
threadPrint	<code>state_queue: ThreadQueue*, format: char*, ...: ?</code>	N/A	Send a print message through the state queue.
tprint	N/A	N/A	Send print message through state queue. Acts like normal printf()
threadExcept	<code>state_queue: ThreadQueue*, exception: uint</code>	N/A	Send an exception message to the state queue.
tekEngineCreateBody	<code>state_queue: ThreadQueue*, bodies: Vector*, object_id: uint, mesh_filename: char*, material_filename: char*, mass: float, friction: float, restitution:</code>	N/A	Create a body given the mesh, material, position etc. NOTE: Will create both a body and a corresponding entity on the graphics thread. The object ids are assigned in order, filling gaps in the order when they appear.

	float, position: vec3, rotation: vec4, scale: vec3		
tekEngineUpdateBody	state_queue: ThreadQueue*, bodies: Vector*, object_id: uint, position: vec3, rotation: vec4, scale: vec3	N/A	Update the position and rotation of a body given its object id.
tekEngineDeleteBody	state_queue: ThreadQueue*, bodies: Vector*, object_id: uint	N/A	Delete a body, freeing the object id for reuse and removing the counterpart on the graphics thread.
tekEngineDeleteAllBodies	state_queue: ThreadQueue*, bodies: Vector*	N/A	Delete all non null bodies in the bodies vector.
tekPushInspectState	state_queue: ThreadQueue*, time: float, position: vec3, velocity: vec3	N/A	Push an inspect state to the state queue. This gives the information about the body currently being inspected by the debug menu.
threadThrow	N/A	N/A	Call exception, push exception to state queue and goto cleanup
threadChainThrow	N/A	N/A	Call an exception, push to state queue and goto cleanup
tekEngine	args: void*	N/A	The main physics thread procedure, will run in parallel to the graphics thread. Responsible for logic, has a loop running at fixed time interval.
tekInitEngine	event_queue: ThreadQueue*, state_queue: ThreadQueue*, phys_period: double, thread: unsigned long long*	N/A	Start the physics thread, providing two thread queues to send and recieve events / states.

tekAwaitEngineS top	thread: unsigned long long	N/A	Wait for the engine thread to join.
------------------------	-------------------------------	-----	-------------------------------------

Geometry

Name	Parameters	Return	Description
tekInitGeometry	N/A	N/A	Called when program begins, initialise some things for geometry.
sumVec3VA	dest: vec3, ...: ?	N/A	Find the sum of a number of vec3s. NOTE: Stops counting when a null pointer is reached. Doesn't check if values are actually vec3s.
tetrahedronSignedVolume	point_a: vec3, point_b: vec3, point_c: vec3, point_d: vec3	The signed volume of the tetrahedron (e.g. can be either positive or negative volume): float	Calculate the signed volume of a tetrahedron from its four vertices.
mat3OuterProduct	a: vec3, b: vec3, m: mat3	N/A	Find the outer product of two vectors, $a \otimes b$
mat3Add	a: mat3, b: mat3, m: mat3	N/A	Find the sum of two matrices. NOTE: Acceptable for a or b to be the same matrix as m.
mat3Subtract	a: mat3, b: mat3, m: mat3	N/A	Find the difference between two matrices. NOTE: Acceptable for a or b to be the same matrix as m.
momentOfInertia	a: vec4, b: vec4, mass: float	The moment of inertia: float	Calculate the moment of inertia of a tetrahedron, these make up the leading diagonal of a inertia tensor matrix.
productOfInertia	a: vec4, b: vec4, mass: float	The "product of inertia": float	Helper function to calculate "products of inertia" - used as elements in the inertia tensor.
tetrahedronInertiaTensor	point_a: vec3, point_b: vec3, point_c: vec3, point_d: vec3, mass: float, tensor: mat3	N/A	Calculate the inertia tensor of a tetrahedron based on the four points that make it up.

translateInertiaTensor	tensor: mat3, mass: float, translate: vec3	N/A	Translate an inertia tensor.
scalarTripleProduct	vector_a: vec3, vector_b: vec3, vector_c: vec3	A floating point number, which is the scalar triple product: float	Calculate the scalar triple product of three vectors = cross product of b and c, dotted against a.
triangleNormal	triangle[3]: vec3, normal: vec3	N/A	Get the normal vector given the three points of a triangle.
randomFloat	min: float, max: float	A random number in between min and max: float	Generate a random floating point number within two bounds, min and max.

Body

Name	Parameters	Return	Description
tekCalculateBodyProperties	body: TekBody*	N/A	Calculate the volume, centre of mass and inverse inertia tensor of an object.
tekBodyUpdateTransform	body: TekBody*	N/A	Update the transformation matrix of a body based on its current position and rotation.
tekCreateBody	mesh_filename: char*, mass: float, friction: float, restitution: float, position: vec3, rotation: vec4, scale: vec3, body: TekBody*	N/A	Create an instance of a body given an empty TekBody struct. NOTE: Will calculate properties of the body given the mesh data, so could take time for larger objects. Will also allocate memory to store vertices.
tekBodyAdvanceTime	body: TekBody*, delta_time: float, gravity: float	N/A	Simulate the effect of a certain amount of time passing on the body's position and rotation. NOTE: Simulate linearly, e.g. with constant acceleration between the two points in time.
tekBodyApplyImpulse	body: TekBody*, point_of_application: vec3,	N/A	Apply an impulse (change in momentum) to a body. NOTE: The point of application is in world

	impulse: vec3, delta_time: float		coordinates, not relative to the body.
tekBodySetMass	body: TekBody*, mass: float	N/A	Update the mass of a body. Don't set the mass directly because the mass affects the density, inertia tensor and other properties that need to be updated.
tekDeleteBody	body: TekBody*	N/A	Delete a TekBody by freeing the vertices that were allocated.

Collisions

Name	Parameters	Return	Description
tekColliderDelete	N/A	N/A	Called at the end of the program to free any allocated structures.
TekColliderInit	N/A	N/A	Initialise some supporting data structures that are needed by many of the methods used in the collision detection process.
tekCheckOBBCollision	obb_a: OBB*, obb_b: OBB*	1 if there was a collision, 0 otherwise: flag	Check whether there is a collision between two OBBs using the separating axis theorem.
tekCreateOBTransform	obb: OBB*, transform: mat4	N/A	Create a transformation matrix that will convert an OBB into an AABB centred around the origin.
tekCheckAABBTriangleCollision	half_extents[3]: float, triangle[3]: vec3	1 if there was a collision, 0 otherwise: flag	Check for a collision between an AABB and a triangle. AABB is assumed to be centred around the origin, with extends of +/- each half extent in that axis.
tekCheckOBBTriangleCollision	obb: OBB*, triangle[3]: vec3	1 if there was a collision, 0 otherwise: flag	Check for a collision between a triangle and a single OBB. Works by transforming both into a position where the OBB can be treated as an AABB (axis aligned bounding box) to reduce calculation.
tekCheckOBBTrianglesCollision	obb: OBB*, triangles: vec3*, num_triangles: uint	1 if there was a collision between any of the triangles	Check for a collision between an array of triangles and an OBB.

		and the obb, 0 otherwise: flag	
tekCalculateOrientation	triangle[3]: vec3, point: vec3	The orientation of the 4 points: float	Calculate the 'orientation' of a triangle versus a point, it is the scalar triple product of the triangle translated by the position vector of the point.
tekTriangleFurthestPoint	triangle[3]: vec3, direction: vec3	closest_point The index of the closest point on the triangle: uint	Return the furthest point on a triangle in a given search direction.
tekTriangleSupport	triangle_a[3]: vec3, triangle_b[3]: vec3, direction: vec3, point: TekPolytopeVertex*	N/A	Triangle support function. Finds a point of a Minkowski Difference that has the largest magnitude in a specified direction.
tekUpdateLineSimplex	direction: vec3, simplex[4]: TekPolytopeVertex	N/A	Update a simplex which has two points.
tekUpdateTriangleSimplexLineAB	ao: vec3, ab: vec3, direction: vec3, len_simplex: uint*	N/A	Helper function to remove duplicate code for case involving a line segment AB
tekUpdateTriangleSimplex	direction: vec3, simplex[4]: TekPolytopeVertex, len_simplex: uint*	N/A	Update a simplex which has three points. The final simplex could have between 1 and 3 points, and direction will be changed based on features of simplex.
tekUpdateTetrahedronSimplex	direction: vec3, simplex[4]: TekPolytopeVertex, len_simplex: uint*	1 if the origin is contained, 0 if not: int	Update a simplex with 4 vertices. Tests each valid face as a separate triangle, and determines if origin is contained.
tekUpdateSimplex	direction: vec3, simplex[4]: TekPolytopeVertex, len_simplex:	1 if the origin is contained within the simplex, 0 if	Call the correct update function based on the number of vertices in the simplex.

	uint*	not or if the simplex is not a tetrahedron: int	
tekCheckTriangleCollision	triangle_a[3]: vec3, triangle_b[3]: vec3, simplex[4]: TekPolytopeVert ex, len_simplex: uint*, separation: float*	Whether or not the triangles are colliding (0 if not, 1 if they are): int	Check for a collision between two triangles using GJK algorithm
tekTriangleTest	triangle_a[3]: vec3, triangle_b[3]: vec3	1 if there was a collision between them, 0 otherwise. (I presume): int	Old test function that survived because I forgot to remove it. But now its too late.
tekGetMinAxis	vector: vec3	0 if X is the smallest, 1 if Y is the smallest and 2 if Z is the smallest: uint	Return the axis of smallest magnitude of a vector.
tekGrowSimplex	triangle_a[3]: vec3, triangle_b[3]: vec3, simplex[4]: TekPolytopeVert ex, len_simplex: uint	N/A	Blow up a simplex so that it is 3D - some simplexes can end when they are a line or a triangle, but the EPA algorithm requires that there is a tetrahedron.
tekGetClosestFace	vertices: Vector*, faces: Vector*, face_index: uint*, face_distance: float*, face_normal: vec3	N/A	Find the closest face of a polytope to the origin.
tekRemoveEdgeFromPolytope	edges: Vector*, edges_bitset: BitSet*, indices[2]: uint	N/A	Remove an edge from a polytope given the indices of the two vertices that make up that edge.
tekRemoveFace	faces: Vector*,	N/A	Remove a face from a polytope,

FromPolytope	edges: Vector*, edges_bitset: BitSet*, face_index: uint		and record any edges that could possibly be removed in the edges vector, and whether or not they should actually be removed in the edges bitset.
tekRemoveAllVisibleFaces	vertices: Vector*, faces: Vector*, edges: Vector*, edges_bitset: BitSet*, support: vec3	N/A	Iterate over faces and remove any which have the support in their positive halfspace
tekAddFaceToPolytope	faces: Vector*, index_a: uint, index_b: uint, index_c: uint	N/A	Add a face (3 vertex indices) to a vector filled with faces. (face_buffer) NOTE: Faces are treated with counter-clockwise winding order. So the "top" of the triangle is the direction where a->b, b->c, c->a is an anticlockwise order.
tekAddFillerFace	vertices: Vector*, faces: Vector*, edge_indices[2]: uint, support_index: uint	N/A	Add a face to fill a gap formed by adding a new support point.
tekAddAllFillerFaces	vertices: Vector*, faces: Vector*, edges: Vector*, edges_bitset: BitSet*, support_index: uint	N/A	If a point is added to a polytope, it produces a "hole" in the polytope which needs to be filled, which this function does.
tekPrintPolytope	vertices: Vector*, faces: Vector*	N/A	Print out a polytope, this is a collection of vertices in face groups that form a closed shape / minkowski sum.
tekProjectOriginToBarycentric	point_a: vec3, point_b: vec3, point_c: vec3, barycentric: vec3	N/A	Take a triangle, and project the origin in the direction of the triangle's face normal. Then get the barycentric coordinates of where the projection meets the triangle.

tekCreatePointFromBarycentric	point_a: vec3, point_b: vec3, point_c: vec3, barycentric: vec3, point: vec3	N/A	Take a triangle plus a barycentric coordinates in terms of u, v and w, and produce a single point that is specified by the barycentric coordinate.
tekGetTriangleCollisionPoints	triangle_a[3]: vec3, triangle_b[3]: vec3, simplex[4]: TekPolytopeVert ex, contact_depth: float*, contact_normal: vec3, contact_a: vec3, contact_b: vec3	N/A	Find the points of collision between two triangles using the Expanding Polytope Algorithm (EPA). Uses the "waste products" of GJK to begin with, and further processes this to find the contact normal and points.
tekGetTriangleCollisionManifold	triangle_a[3]: vec3, triangle_b[3]: vec3, collision: flag*, manifold: TekCollisionMani fold*	N/A	Generate a collision manifold between two triangles, if they are colliding.
tekCheckTrianglesCollision	triangles_a: vec3*, num_triangles_a: uint, triangles_b: vec3*, num_triangles_b: uint, collision: flag*, manifold: TekCollisionMani fold*	N/A	Check for collisions between two arrays of triangles, and copy collision data into an empty manifold.
tekIsCoordinateEquivalent	point_a: vec3, point_b: vec3	1 if they are very close, 0 otherwise: int	Check if two coordinates are within a tiny tolerance of each other. (less than 1e-6 units separating them)
tekIsManifoldEquivalent	manifold_a: TekCollisionMani fold*, manifold_b: TekCollisionMani fold*	1 if they are equivalent, 0 otherwise: int	Check if two manifolds have a contact point that is within a tiny tolerance of each other.

tekDoesManifoldContainContacts	manifold_vector: Vector*, manifold: TekCollisionManifold*, contained: flag*	N/A	Search the list of contact manifolds and see if a potential new contact already exists.
getChild	N/A	N/A	Helper function to get a child, essentially a mapping of (0,1) -> (node.left,node.right)
tekGetCollisionManifolds	body_a: TekBody*, body_b: TekBody*, collision: flag*, manifold_vector: Vector*	N/A	Get the collision manifolds relating to the two bodies, and add them to a provided vector. Gives information such as contact position, depth, normals, tangent vectors etc.
tekSetupInvMassMatrix	body_a: TekBody*, body_b: TekBody*, inv_mass_matrix [4]: mat3	N/A	Create an inverse mass matrix, kinda a 12x12 matrix, 0,1 = body a inverse mass + inverse inertia tensor, 2,3 = body b ...
tekApplyCollision	body_a: TekBody*, body_b: TekBody*, manifold: TekCollisionManifold*	N/A	Apply collision between two bodies, based on the contact manifold between them.
tekSolveCollisions	bodies: Vector*, phys_period: float	N/A	Decide which bodies are colliding and apply impulses to separate any colliding bodies.

Collider

Name	Parameters	Return	Description
symmetricMatrixCalculateEigenvectors	matrix: mat3, eigenvectors[3]: vec3, eigenvalues[3]: float	N/A	Compute the eigenvectors and eigenvalues of a symmetric matrix. NOTE: Uses LAPACK / ssyev_(...) internally.
solveSimultaneous3Vec3	lhs_vectors[3]: vec3, rhs_vector:	N/A	Solve a simultaneous equation with 3 unknowns and 3 vector

	vec3, result: vec3		equations.
tekCalculateTriangleArea	point_a: vec3, point_b: vec3, point_c: vec3	The area of the triangle formed by those three points: float	Calculate the area of a triangle given three points in space.
tekCheckIndex	N/A	N/A	Ensure that an index is allowed, throw error if not.
tekGenerateTriangleArray	vertices: vec3*, num_vertices: uint, indices: uint*, num_indices: uint, triangles: Vector*	N/A	Fill a Vector with the data for all triangles of a mesh. This means the vertices and area of each triangle. NOTE: 'triangles' is freed if the function fails.
tekCalculateConvexHullMean	triangles: Vector*, indices: uint*, num_indices: uint, mean: vec3	N/A	Calculate the mean point of a set of triangles interpreted as a convex hull.
tekCalculateCovarianceMatrix	triangles: Vector*, indices: uint*, num_indices: uint, mean: vec3, covariance: mat3	N/A	Calculate the covariance matrix of a convex hull. This is a statistical measure of the spread of "matter" in the rigidbody, or like the standard deviation in 3 axes.
tekFindProjectionExtents	triangles: Vector*, indices: uint*, num_indices: uint, axis: vec3, min_p: float*, max_p: float*	N/A	Find the minimum and maximum point of a set of triangles projected onto a single axis. Used to find the half extents of an OBB.
tekCreateOBB	triangles: Vector*, indices: uint*, num_indices: uint, obb: OBB*	N/A	Create an OBB based on a set of triangles. This will be the smallest rectangular prism that contains all points of the triangles.
tekCreateColliderNode	type: flag, id: uint, triangles: Vector*, indices: uint	N/A	Create a collider node struct given some information. Mostly just a helper function to create the

	<code>uint*, num_indices: uint, collider_node: TekColliderNode* *</code>		internal OBB and set some values rather than copy-pasting a 10 line statement.
tekColliderClean up	N/A	N/A	Clean up memory involved in creation of collider.
tekPrintCollider	collider: <code>TekColliderNode* ,</code> indent: uint	N/A	Helper function to print out a collider tree.
tekCreateCollider	body: TekBody*, collider: <code>TekCollider* ,</code>	N/A	Create a collider structure given a body. The body must be initialised and contain the vertex and index data for the mesh.
tekDeleteCollider Node	collider_node: <code>TekColliderNode* ,</code> handedness: flag	N/A	Delete a collider node
tekDeleteCollider	collider: <code>TekCollider* ,</code>	N/A	Delete a collider node by freeing all allocated memory. Will set pointer to NULL to avoid misuse of freed pointer.
tekUpdateOBB	obb: OBB*, transform: mat4	N/A	Update an OBB based on a transformation matrix, so the OBB's coordinates correspond to world coordinates not local coordinates.
tekUpdateLeaf	leaf: <code>TekColliderNode* ,</code> transform: mat4	N/A	Update a leaf node of the collider structure by transforming each vertex by the transform of the object it relates to.

Scenario

Name	Parameters	Return	Description
tekScenarioGetSnapshot	scenario: <code>TekScenario*, snapshot_id: uint, snapshot: TekBodySnapsh</code>	N/A	Get a snapshot from a scenario using its id.

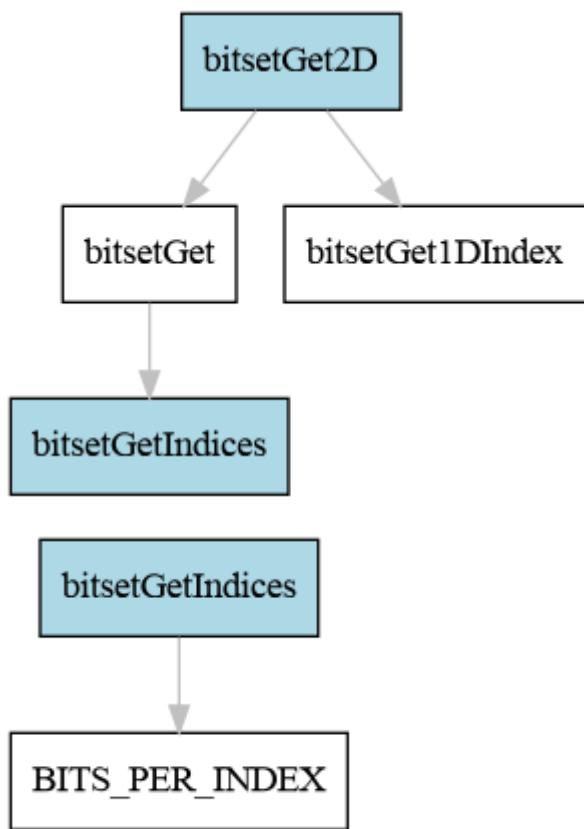
	ot**		
tekScenarioGetByNameIndex	scenario: TekScenario*, name_index: uint, snapshot: TekBodySnapshot**, snapshot_id: int*	N/A	Get a snapshot id/data by using the index at which its name appears in the name list. Very useful in TekGuiList callbacks when the user clicks the name of an object they want and you want the associated snapshot
tekScenarioGetName	scenario: TekScenario*, snapshot_id: uint, snapshot_name: char**	N/A	Get the name of a snapshot using its id. NOTE: The name is a direct pointer into the name list, so don't edit unless you want changes to be reflected visually.
tekScenarioSetName	scenario: TekScenario*, snapshot_id: uint, snapshot_name: char*	N/A	Set the name of a snapshot in a scenario by id. Will copy the name into a new buffer.
tekScenarioGetNextId	scenario: TekScenario*, next_id: uint*	N/A	Get the next available ID in the scenario. Will first check if any ids have been made available by removing old items, else will return the lowest fresh id.
tekScenarioCreatePair	scenario: TekScenario*, copy_snapshot: TekBodySnapshot*, snapshot_id: uint, snapshot_name: char*, pair: TekScenarioPair**	N/A	Create a scenario pair, this includes the data needed for each snapshot such as the name, id and properties.
tekScenarioDeletePair	pair: TekScenarioPair*	N/A	Delete a scenario pair, free allocated memory and the pair.
tekScenarioPutSnapshot	scenario: TekScenario*, copy_snapshot: TekBodySnapshot*, snapshot_id:	N/A	Add a new snapshot or update an existing snapshot with a specified id. Will update the name and data associated with the id.

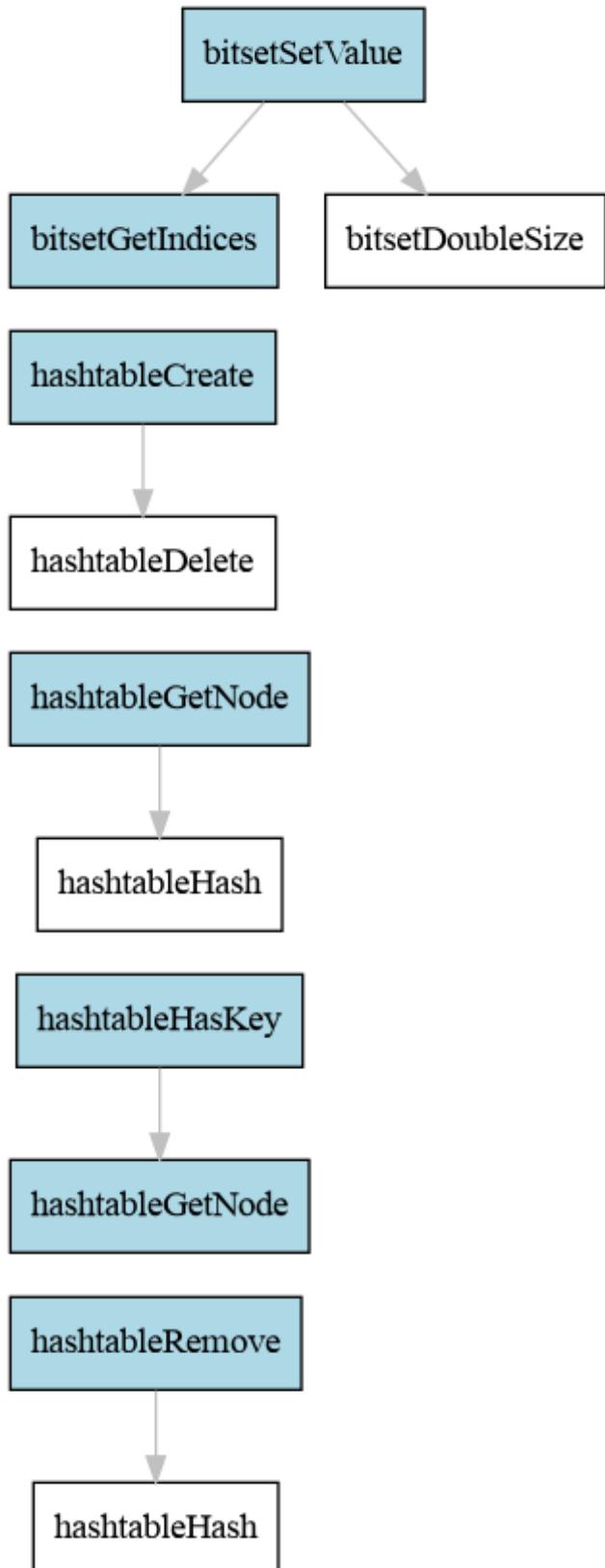
	uint, snapshot_name: char*		
tekScenarioDeleteSnapshot	scenario: TekScenario*, snapshot_id: uint	N/A	Delete a snapshot, freeing any allocated memory of the struct.
tekCreateScenario	scenario: TekScenario*	N/A	Create a new scenario, allocates some memory for different internal structures.
tekScanSnapshot	string: char*, snapshot: TekBodySnapshot*, snapshot_id: uint*, snapshot_name: char*	Number of items successfully scanned: int	Scan a single snapshot from a buffer. Expects the same format as specified by SNAPSHOT_READ_FORMAT - key value pairs of snapshot data seperated by new lines.
tekReadScenarios	scenario_filepath: : char*, scenario: TekScenario*	N/A	Read a scenario from a specified file, and load it into a scenario struct.
tekWriteSnapshot	string: char*, max_length: size_t, snapshot: TekBodySnapshot*, snapshot_id: uint, snapshot_name: char*	bytes written: int	Write a single snapshot of a body to a buffer. Also returns the number of bytes written in the buffer.
tekScenarioGetAllIds	scenario: TekScenario*, ids: uint**, num_ids: uint*	N/A	Allocate a buffer containing the ids of all snapshots in the scenario. NOTE: This function allocates memory which you are responsible for freeing.
tekWriteScenario	scenario: TekScenario*, scenario_filepath: : char*	N/A	Write the data stored in a scenario to a specified filepath.
tekDeleteScenario	scenario: TekScenario*	N/A	Delete a scenario, removing any allocated memory of the struct.

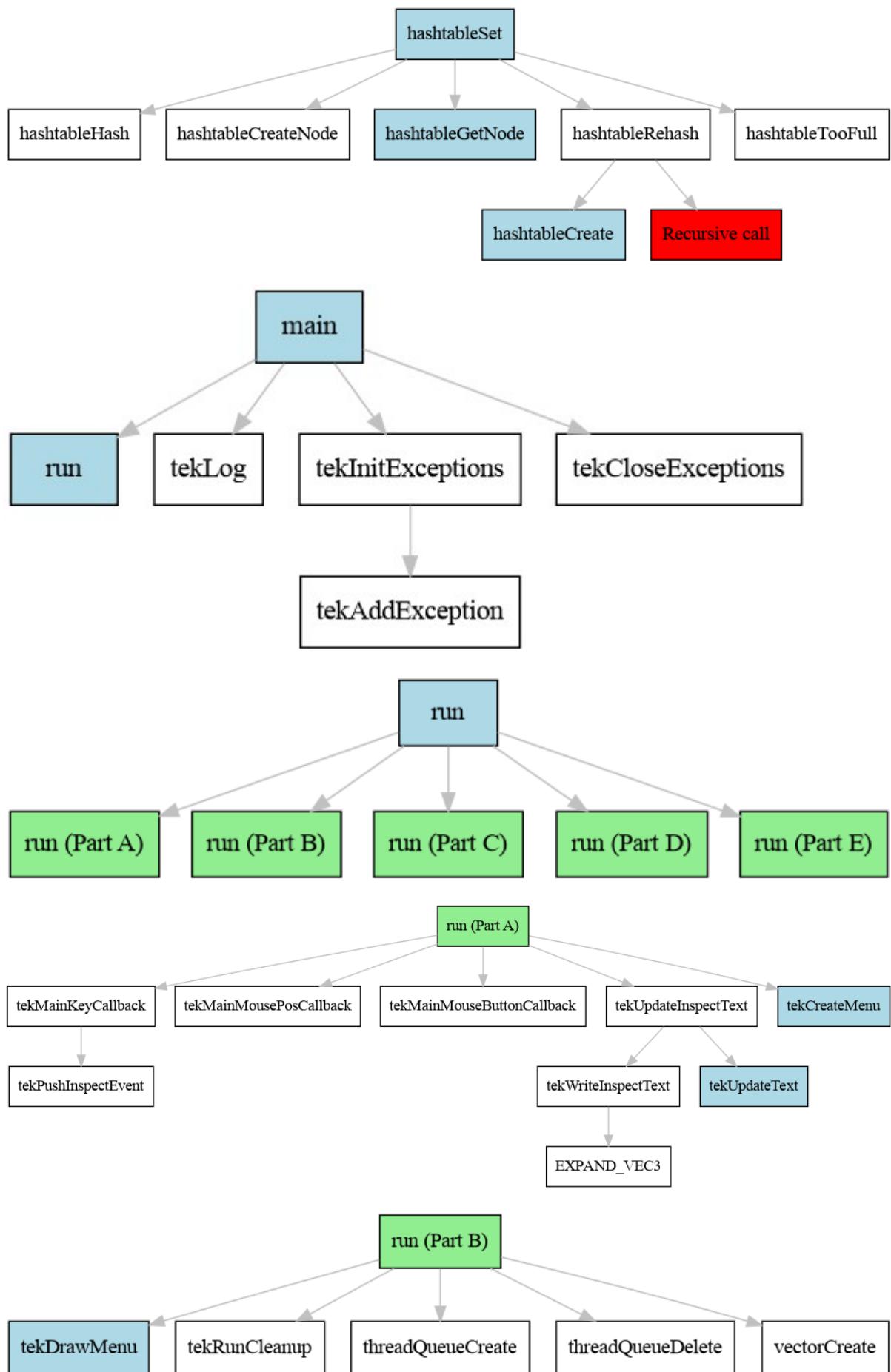
Hierarchy Chart

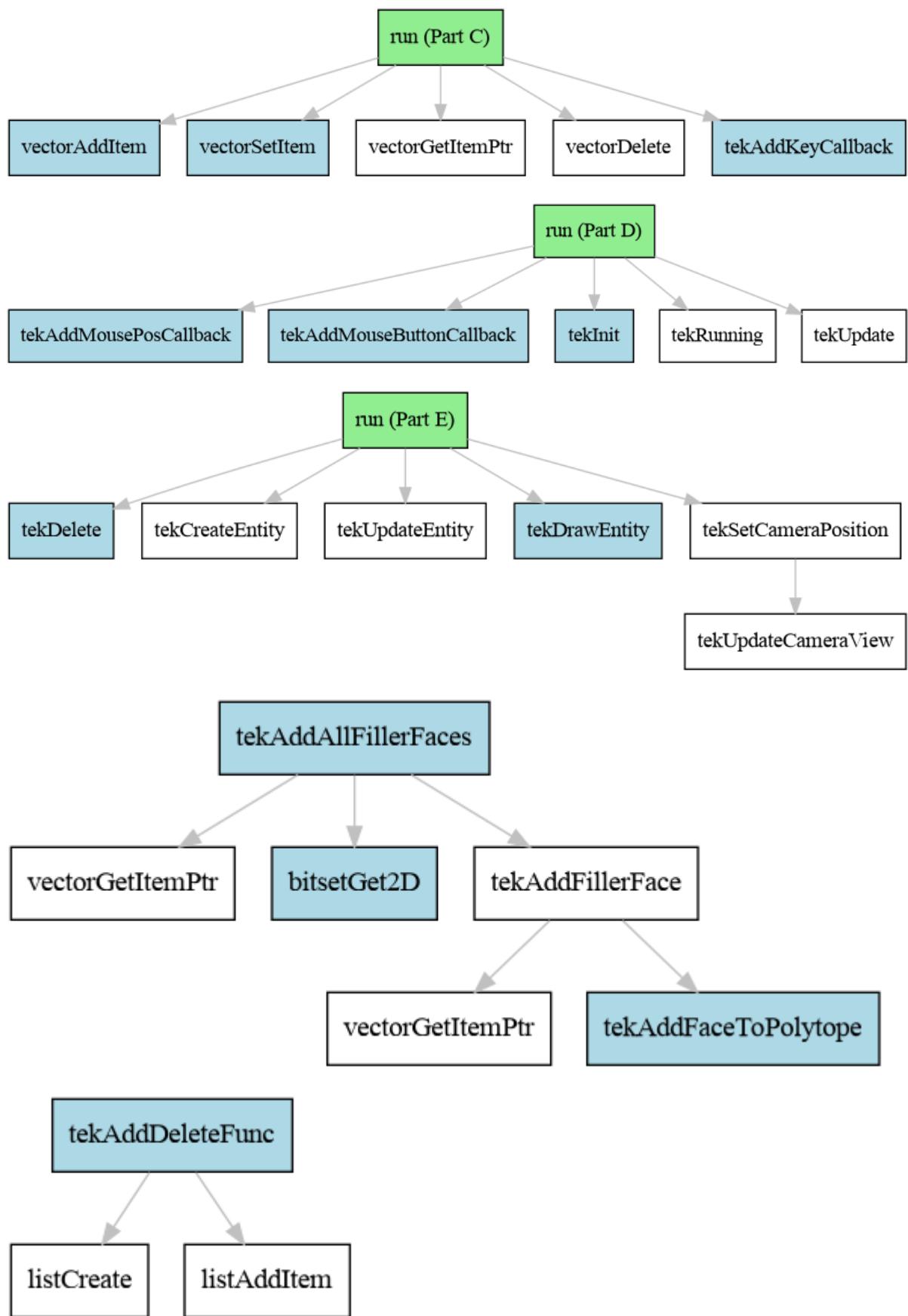
Below is a series of hierarchy charts that link together to make a complete hierarchy chart. The entrypoint of the chart is in the “main” function. The charts have been sorted into alphabetical order by root node. In order to better understand the hierarchy chart, a colour scheme has been implemented.

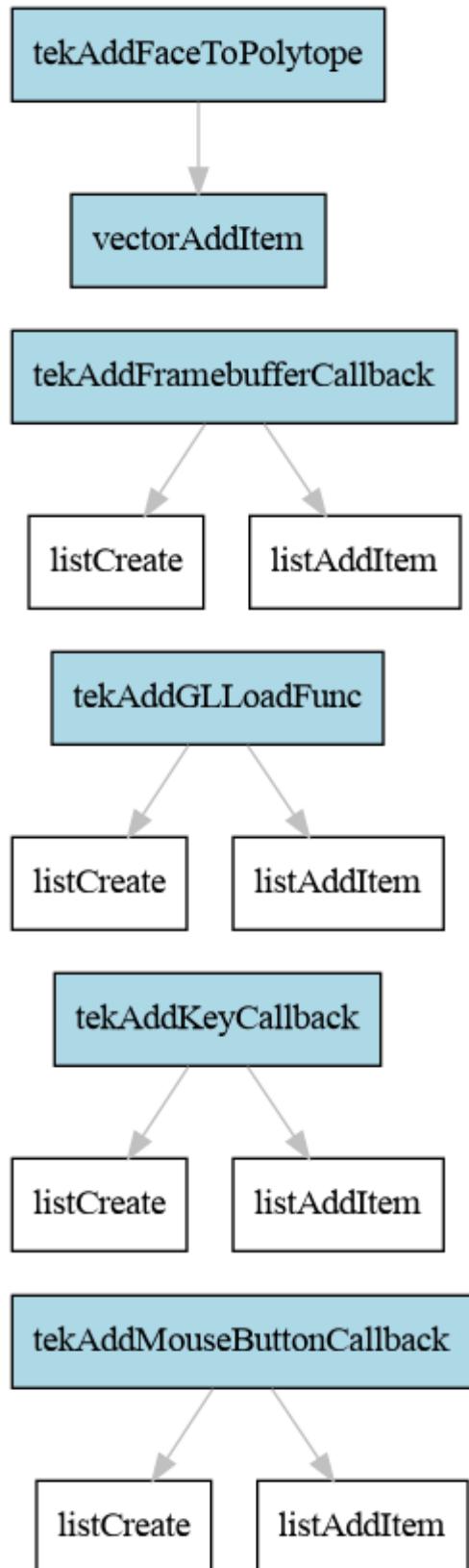
Boxes with a white background represent a function in the code.	Boxes with a blue background represent a function that calls other functions, which has been moved to a separate chart in order to maintain readability.	Boxes with a green background are not functions in the code, it is where a group of sub functions have been collected into a different chart to improve readability.	Boxes with a red background are not functions, they represent that this function is part of a recursive chain. (function calling itself).
---	--	---	--

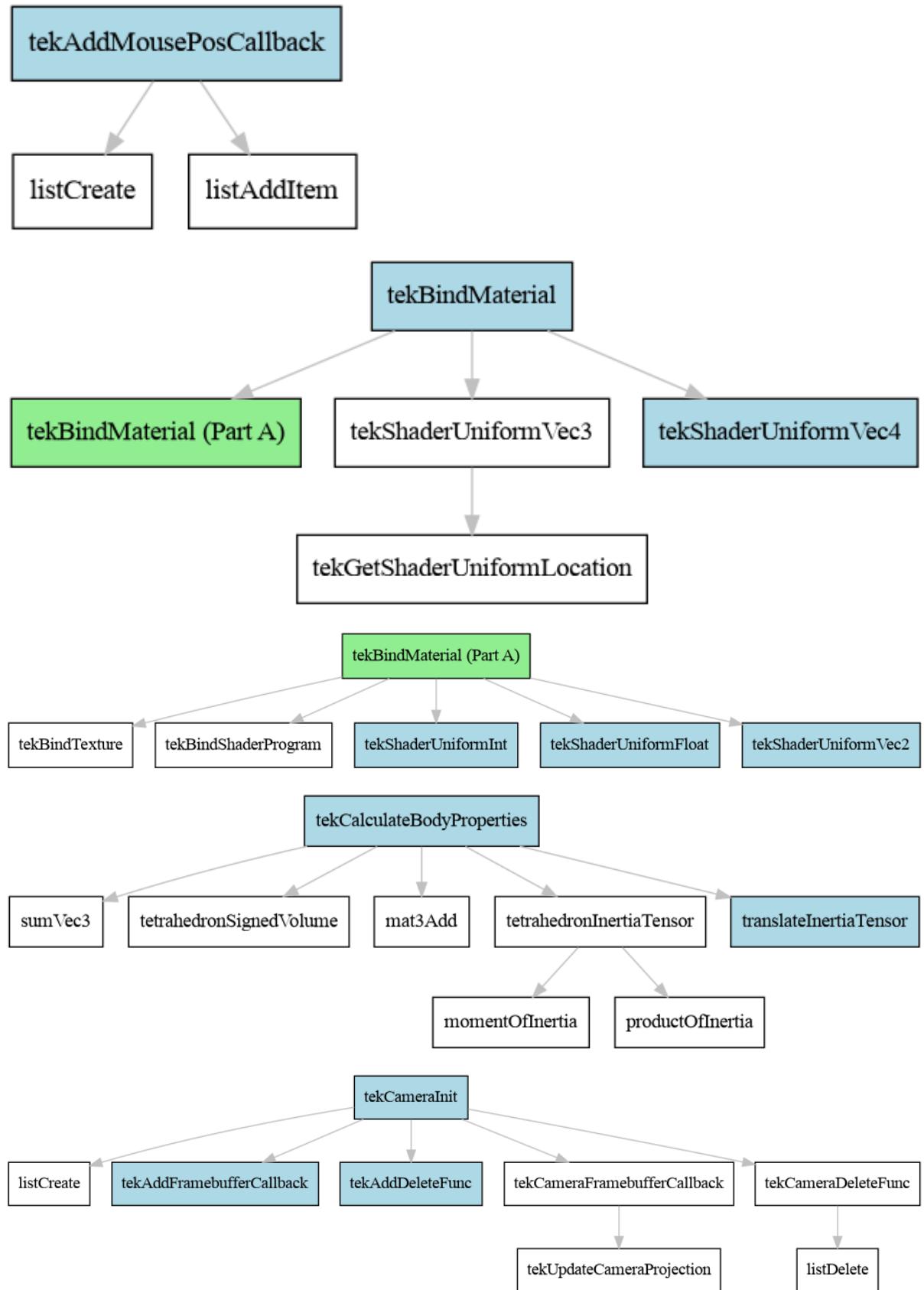


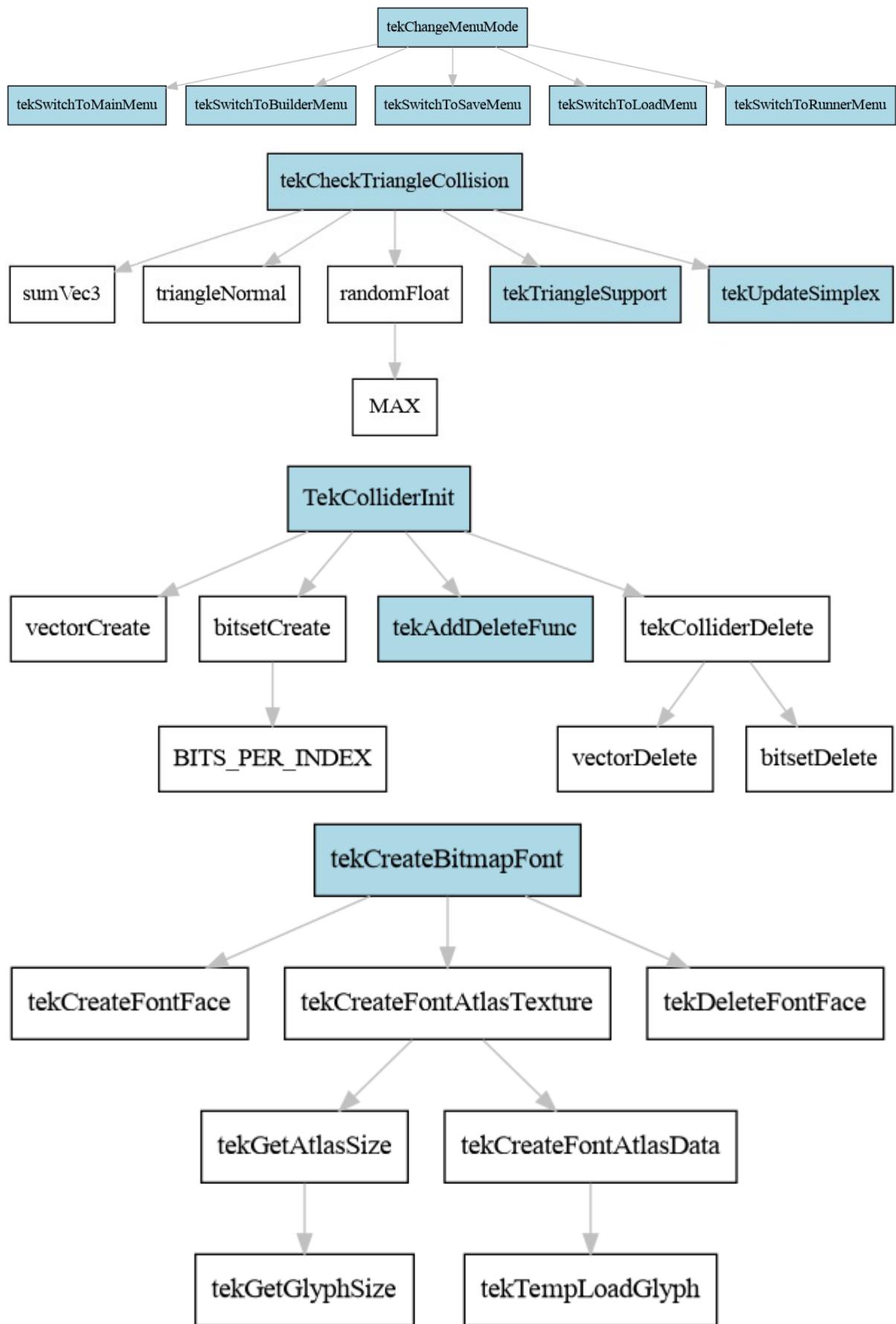


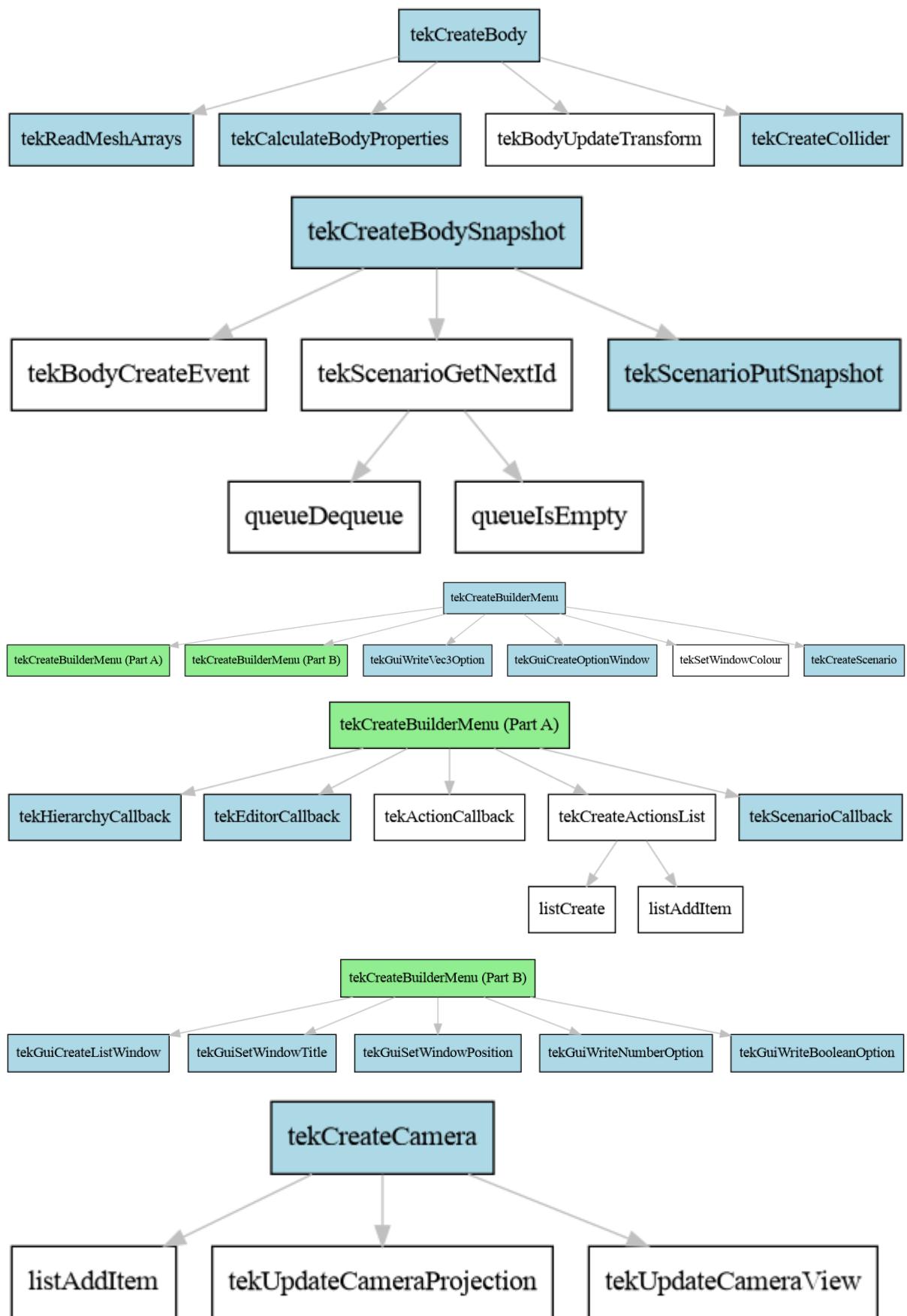


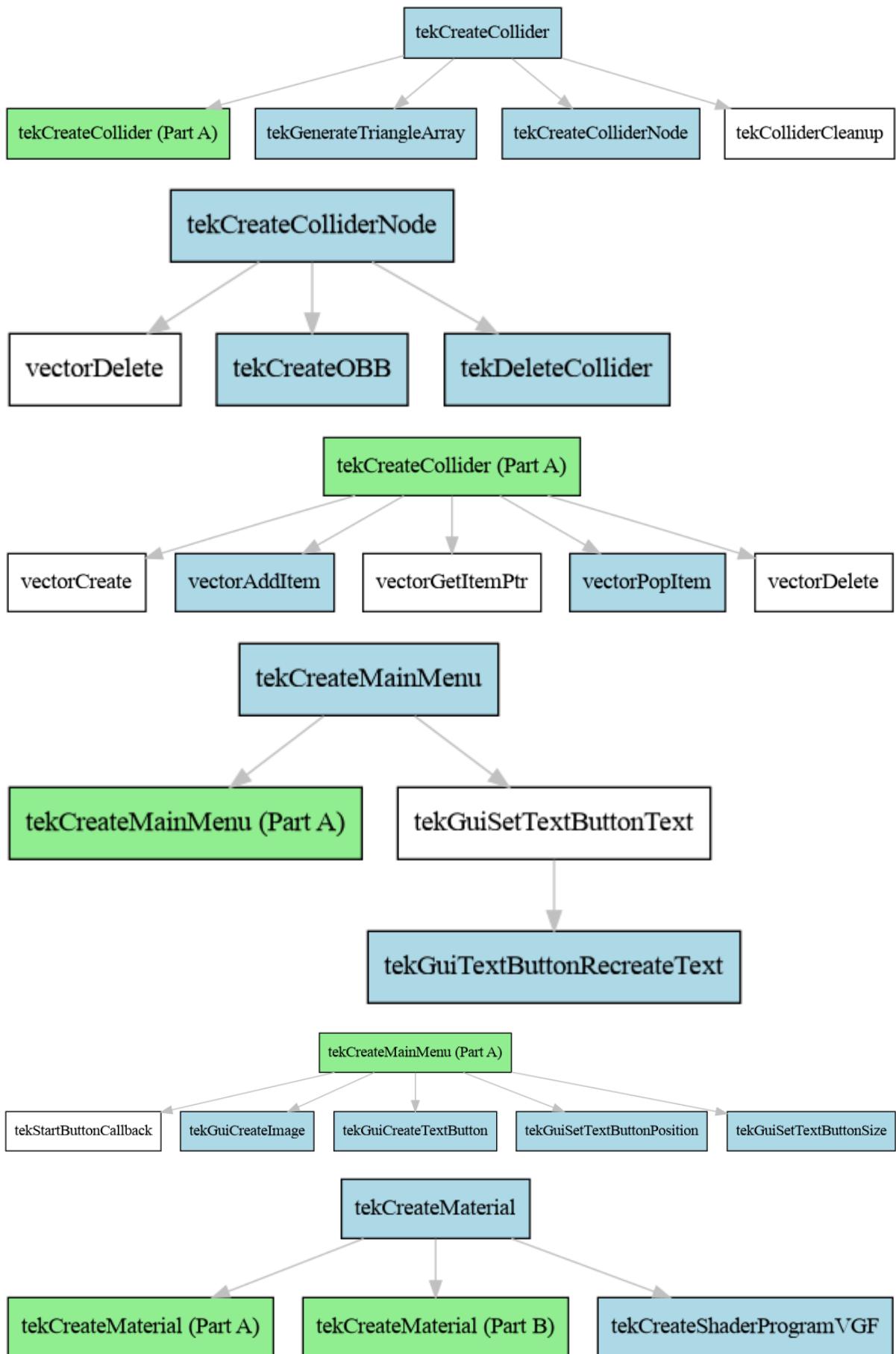


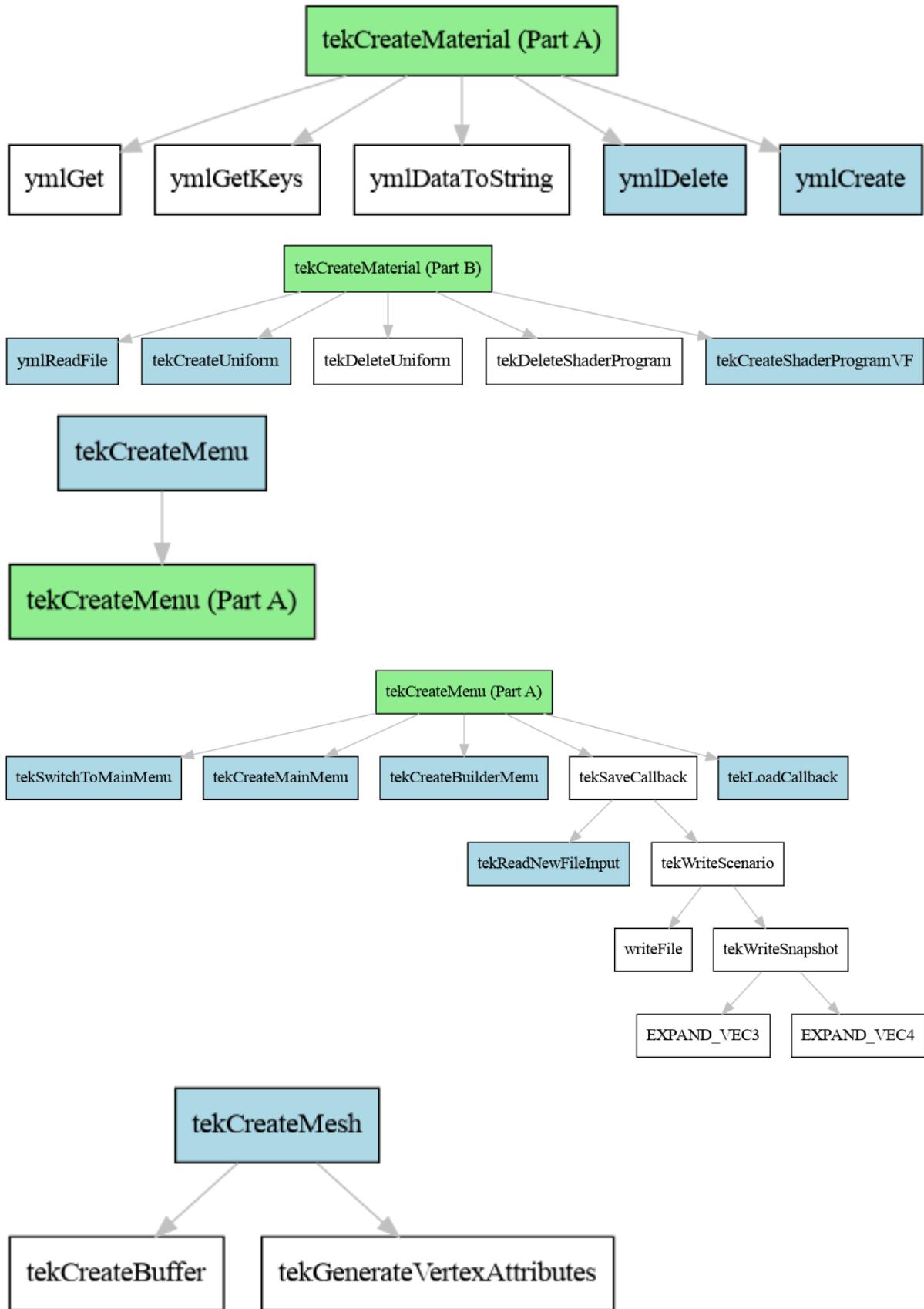


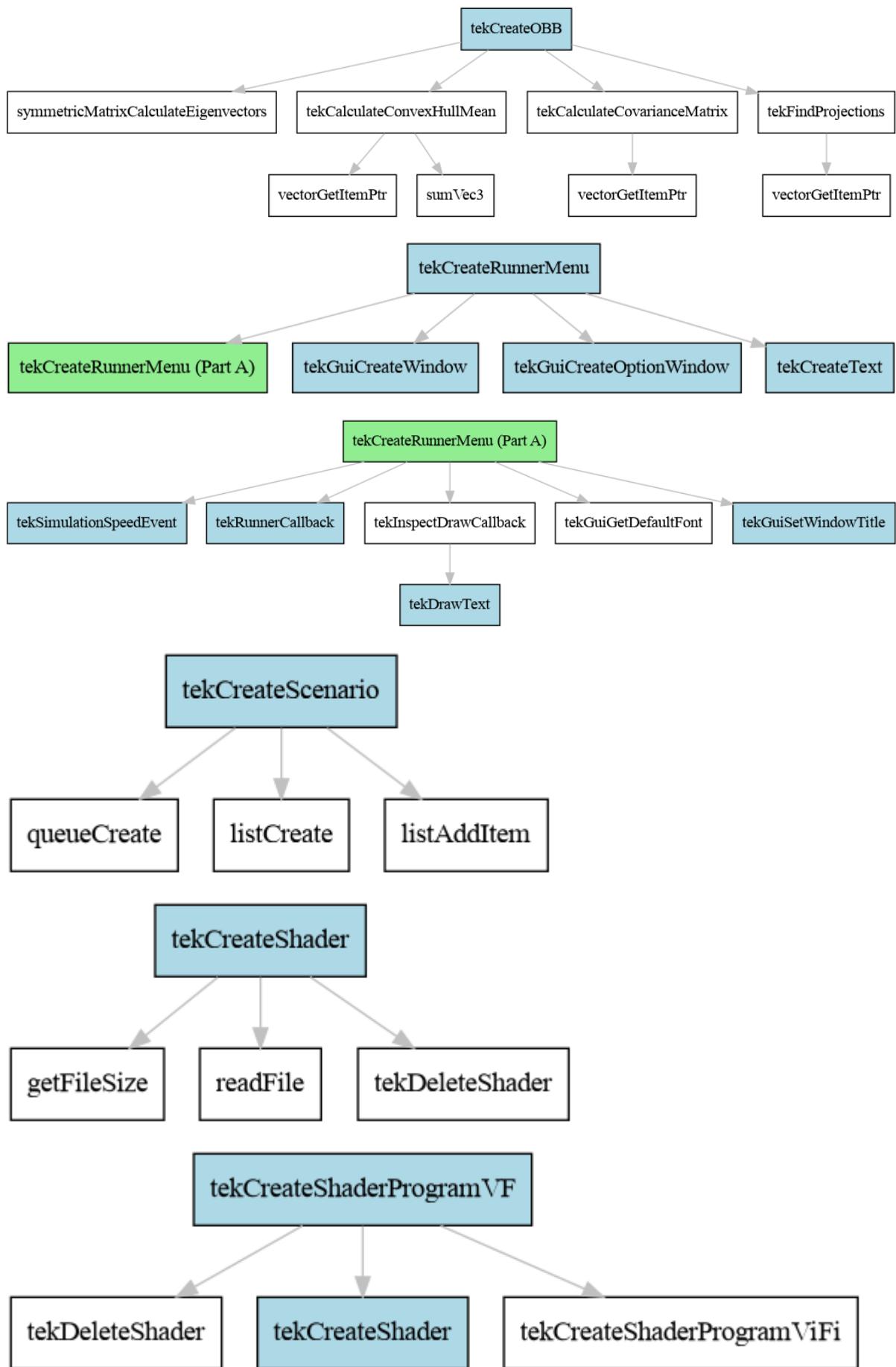


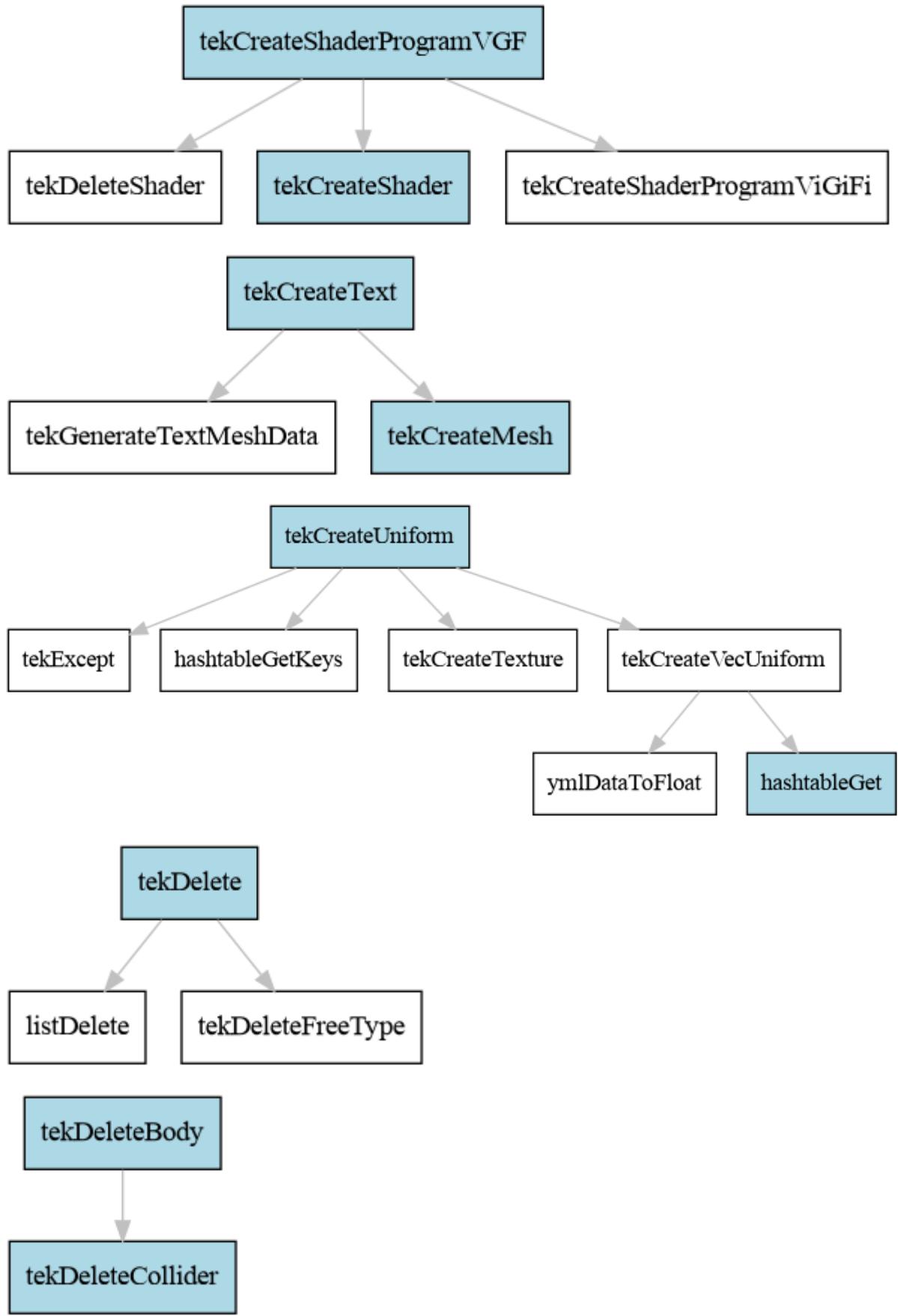


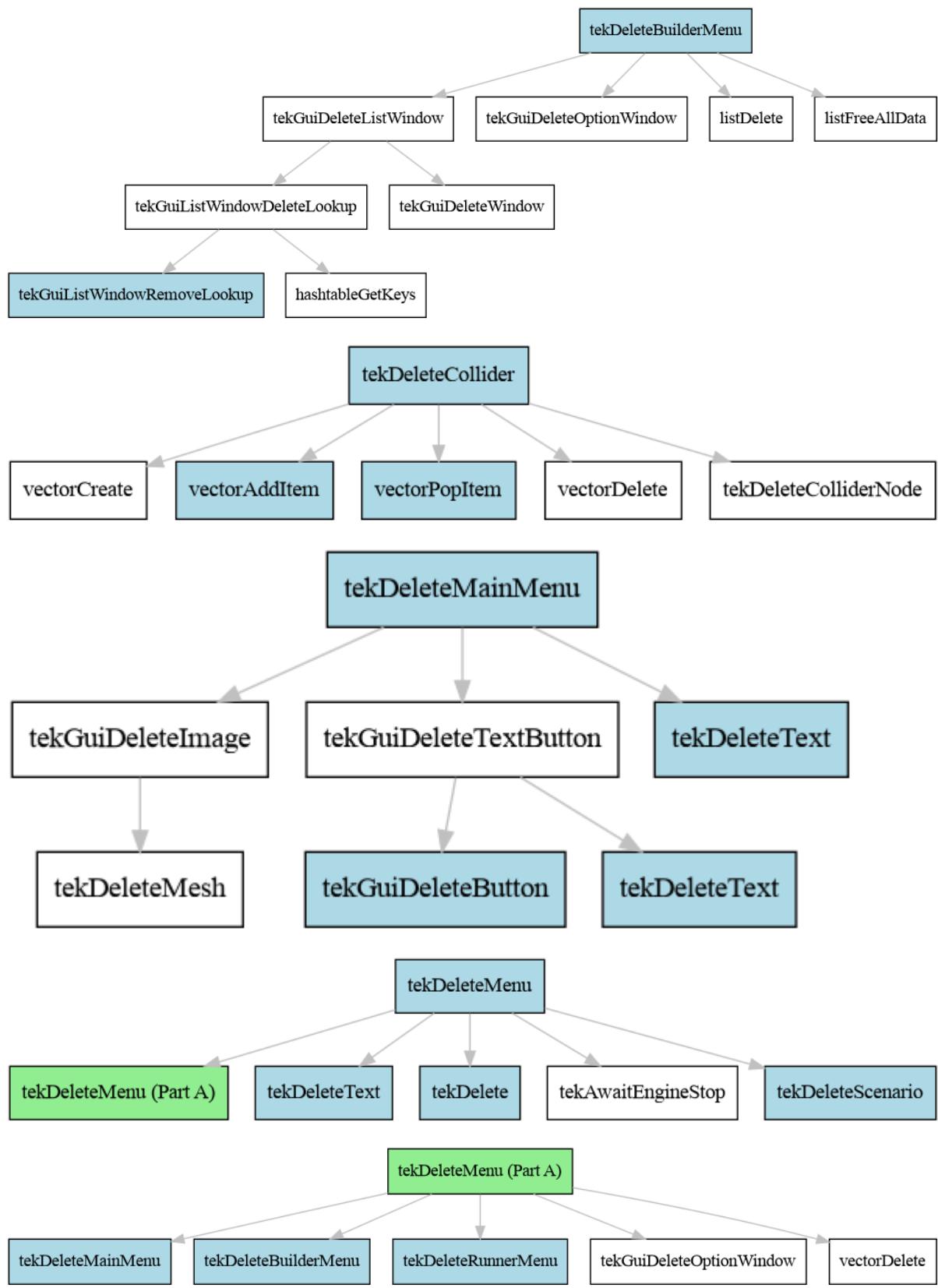


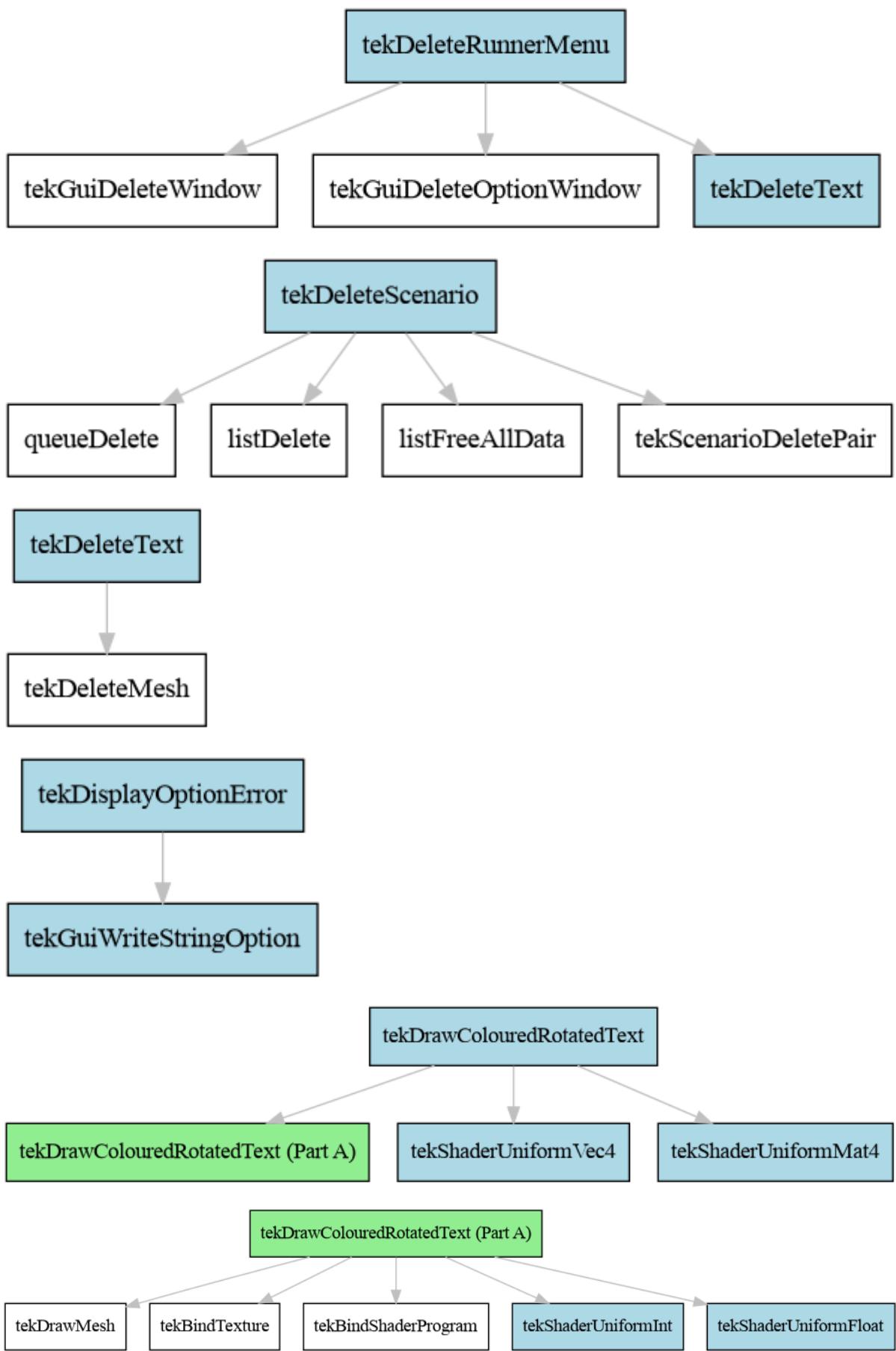


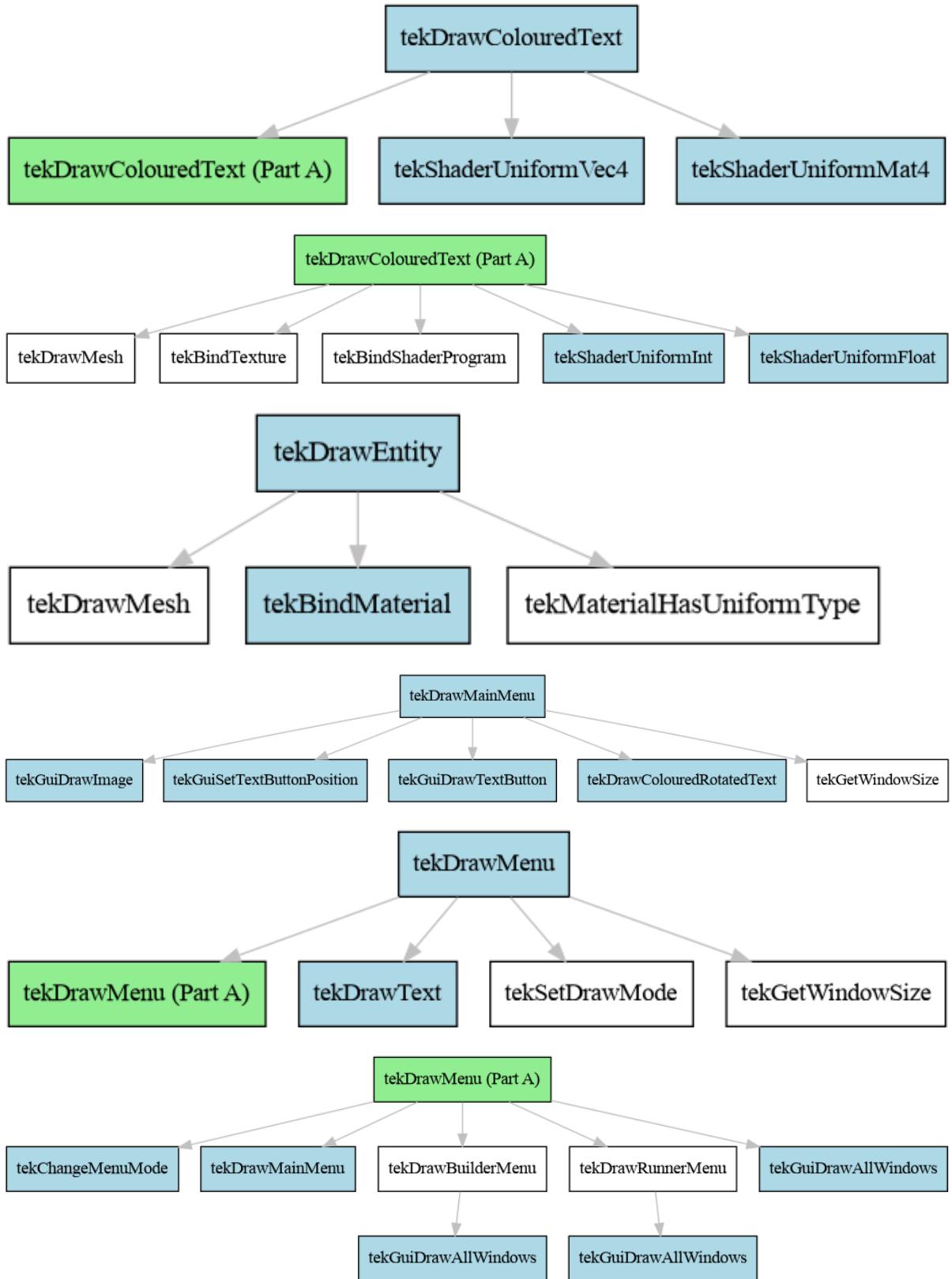


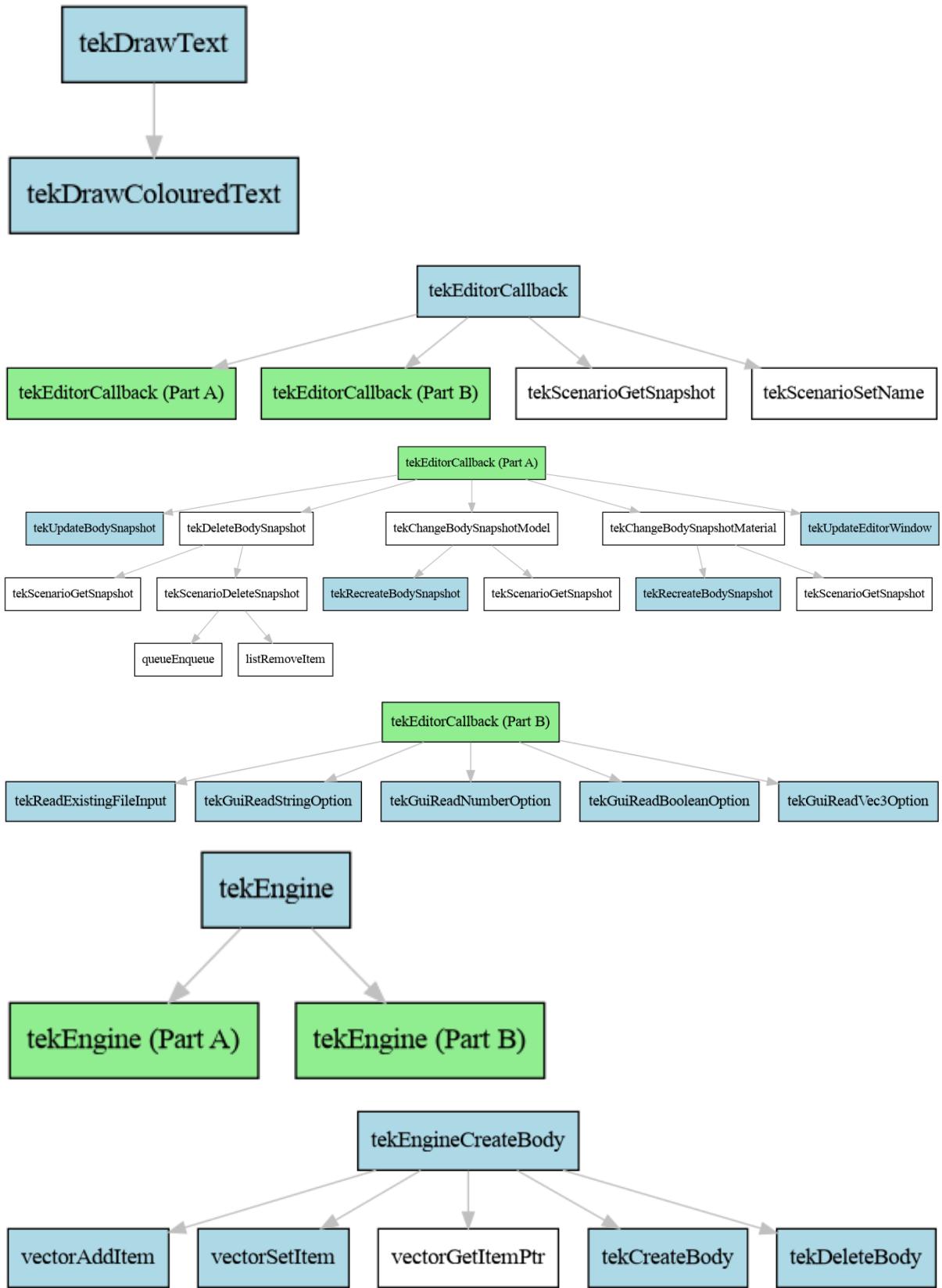


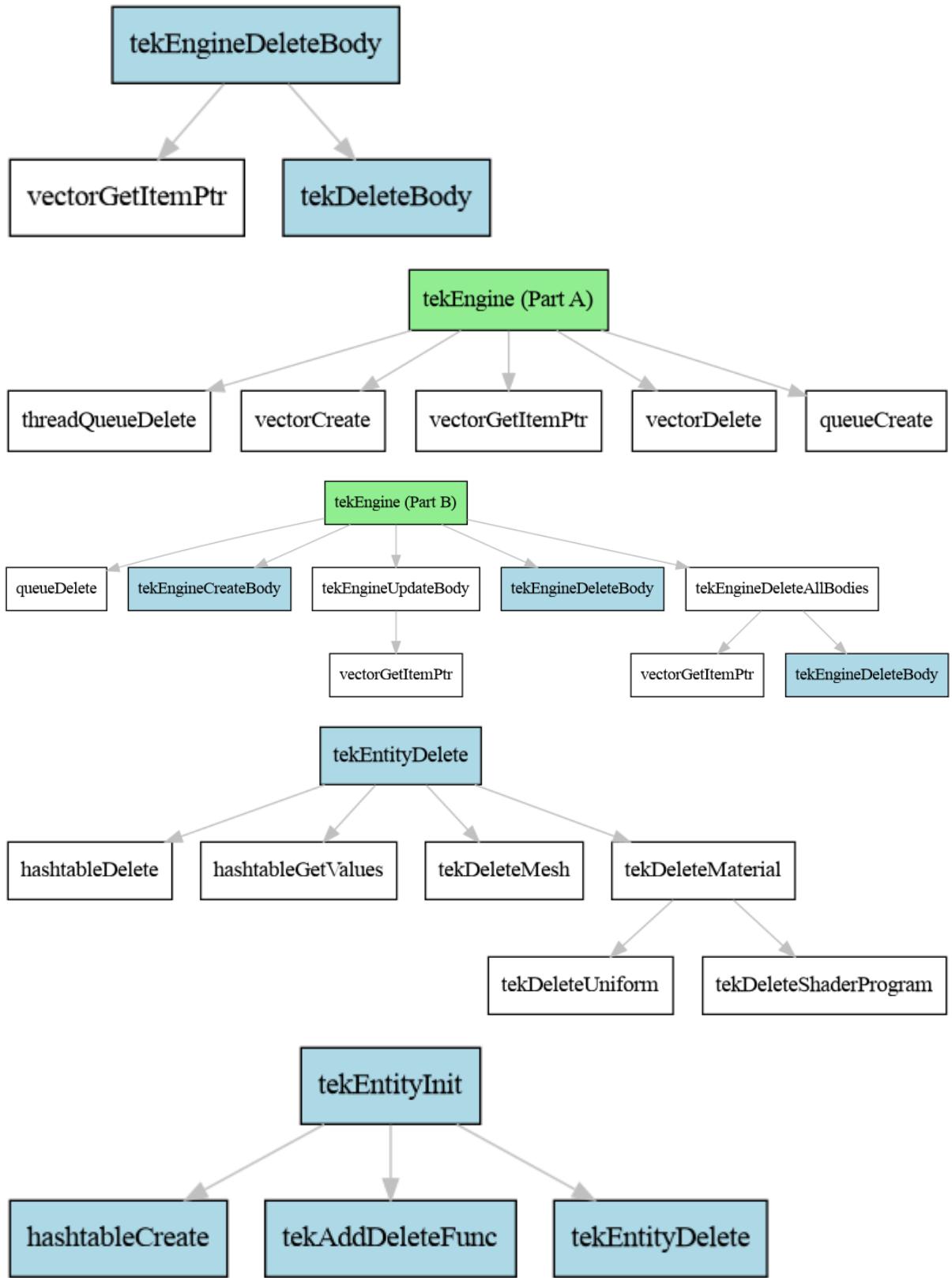


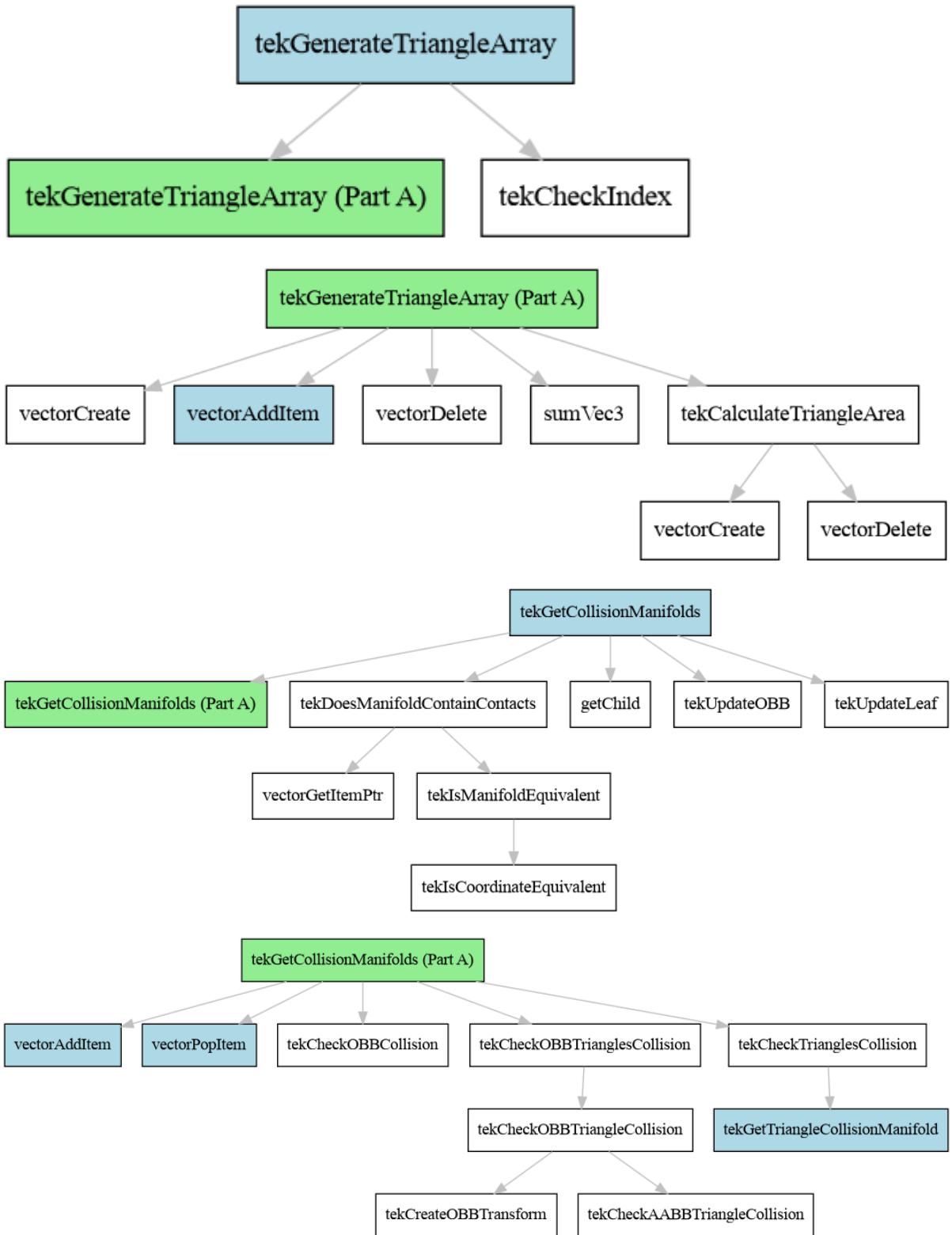


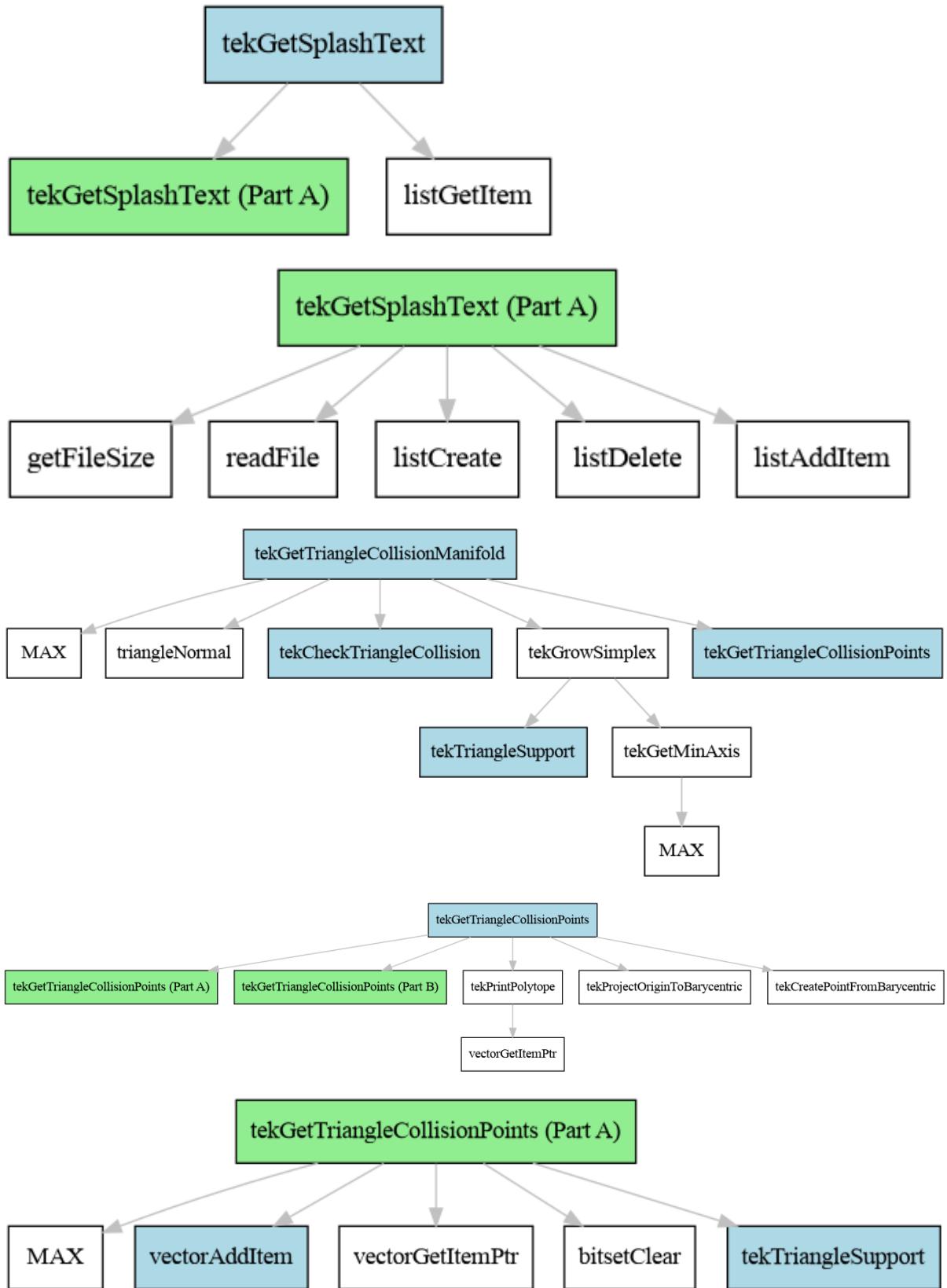


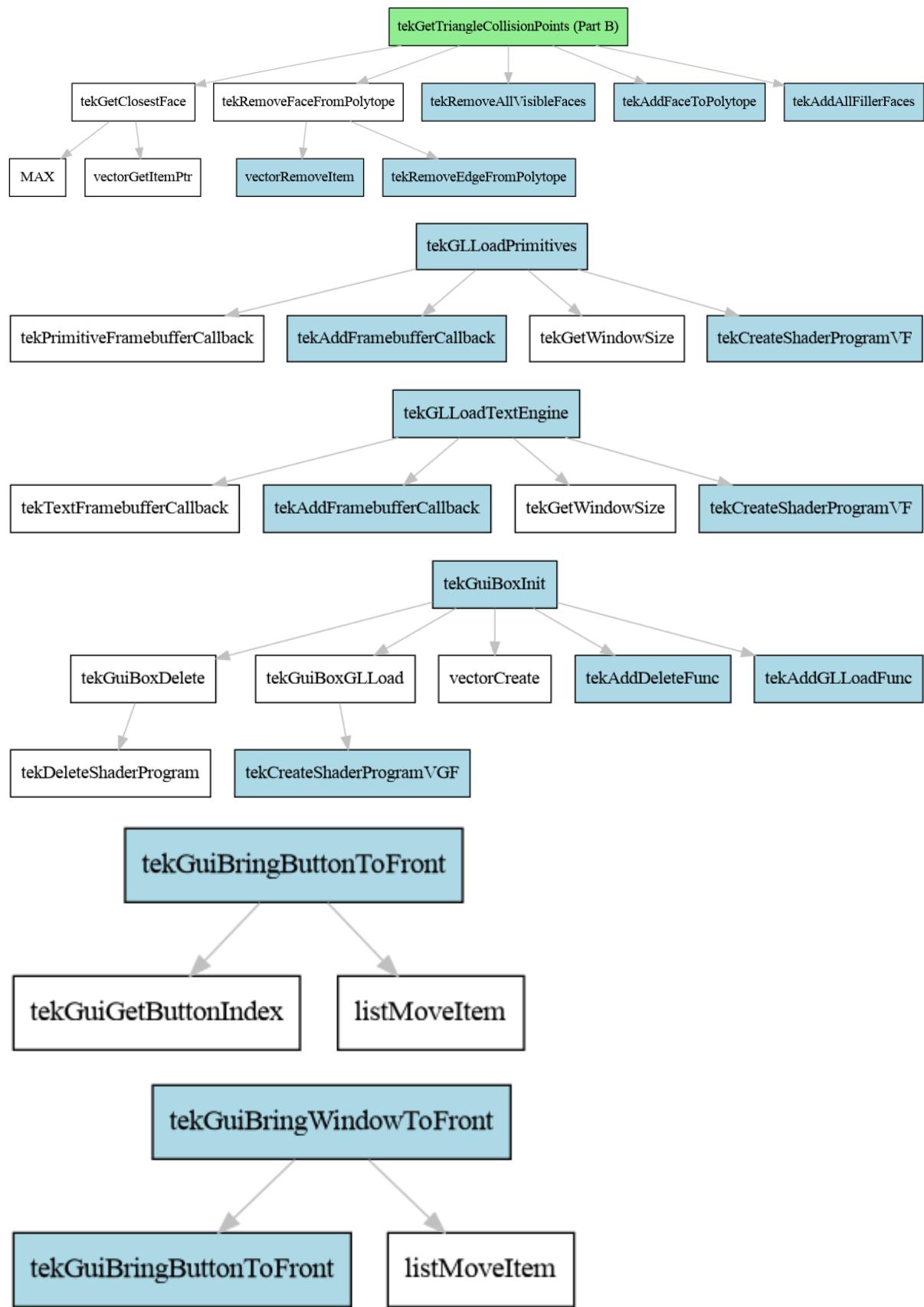




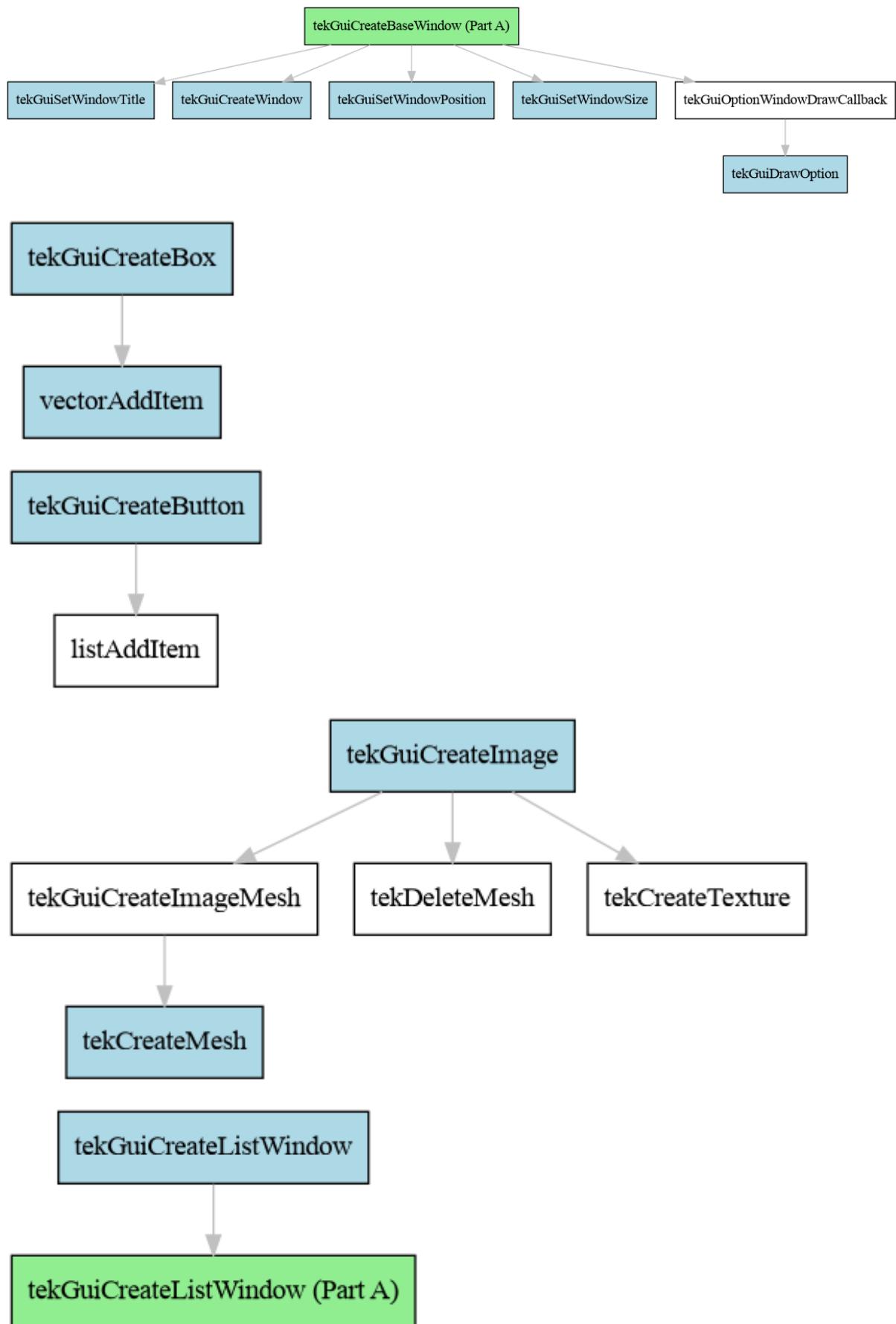


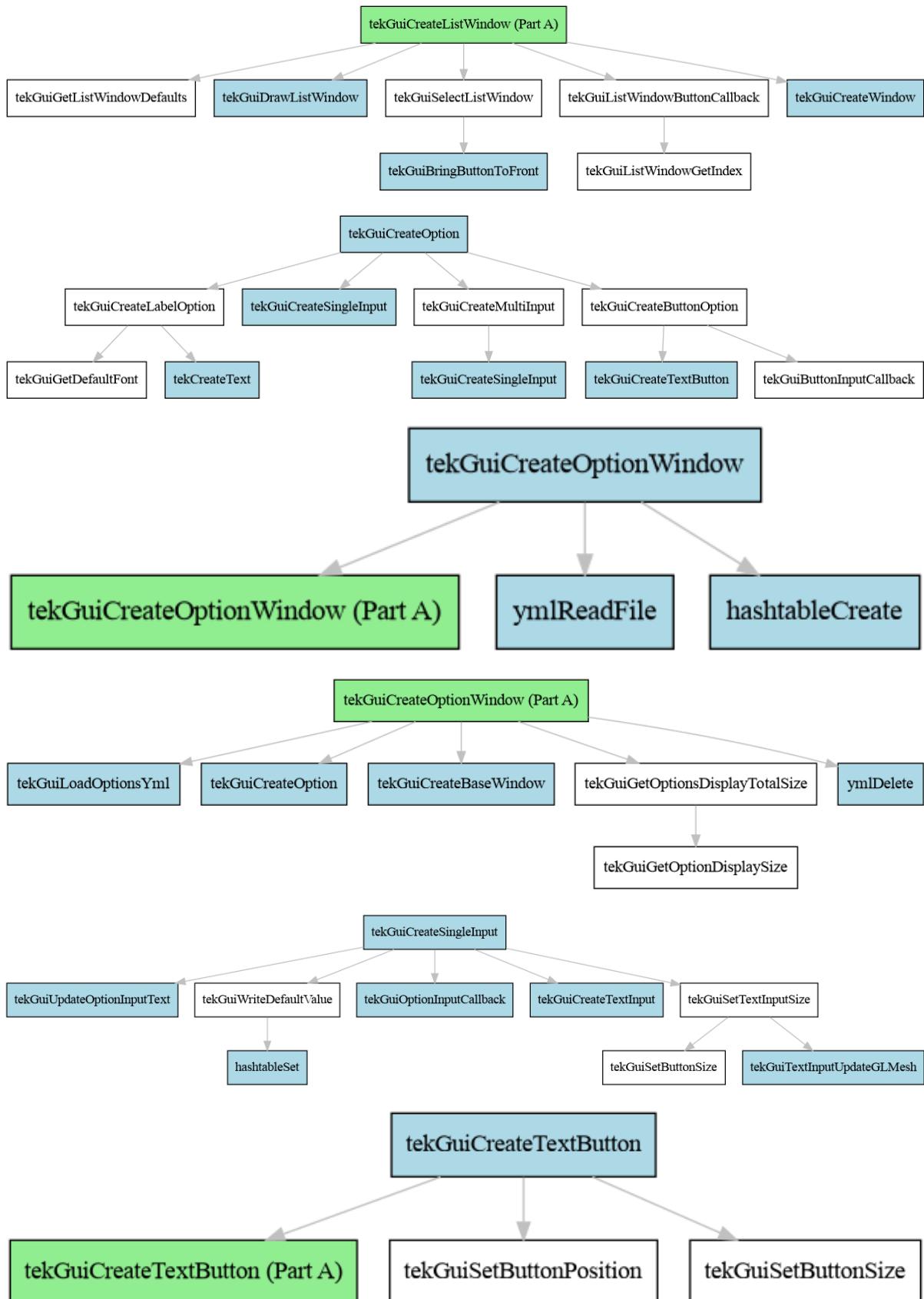


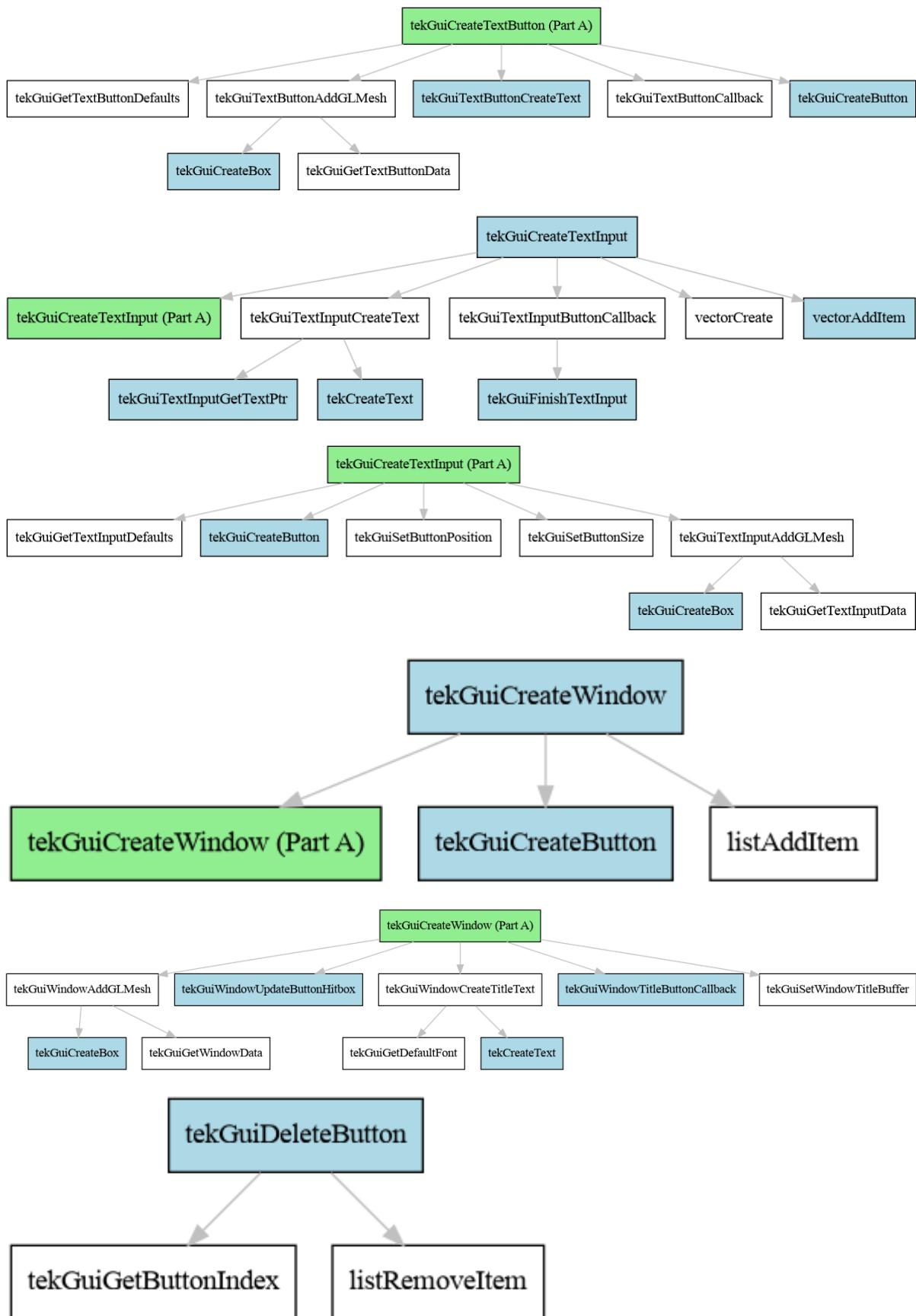


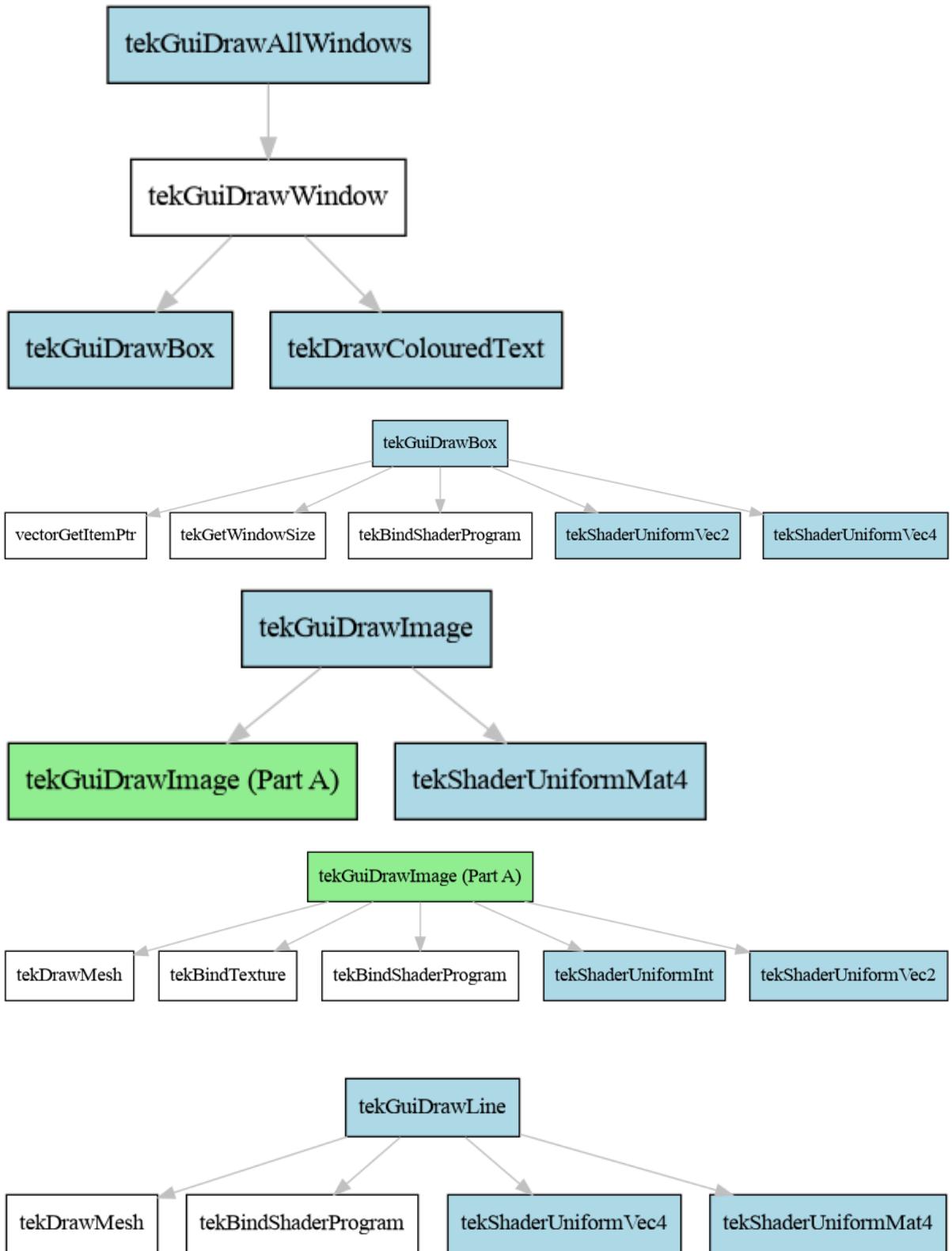


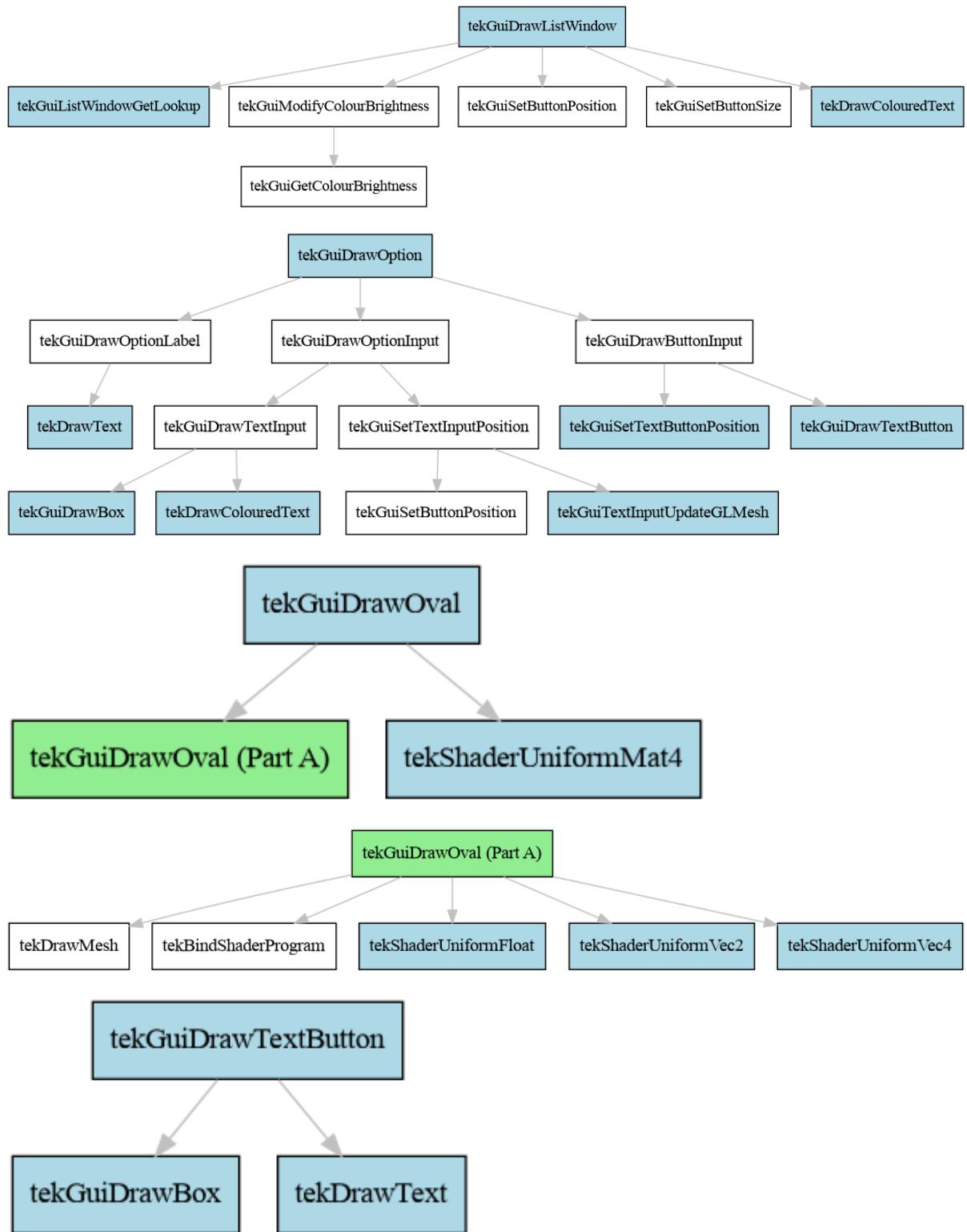


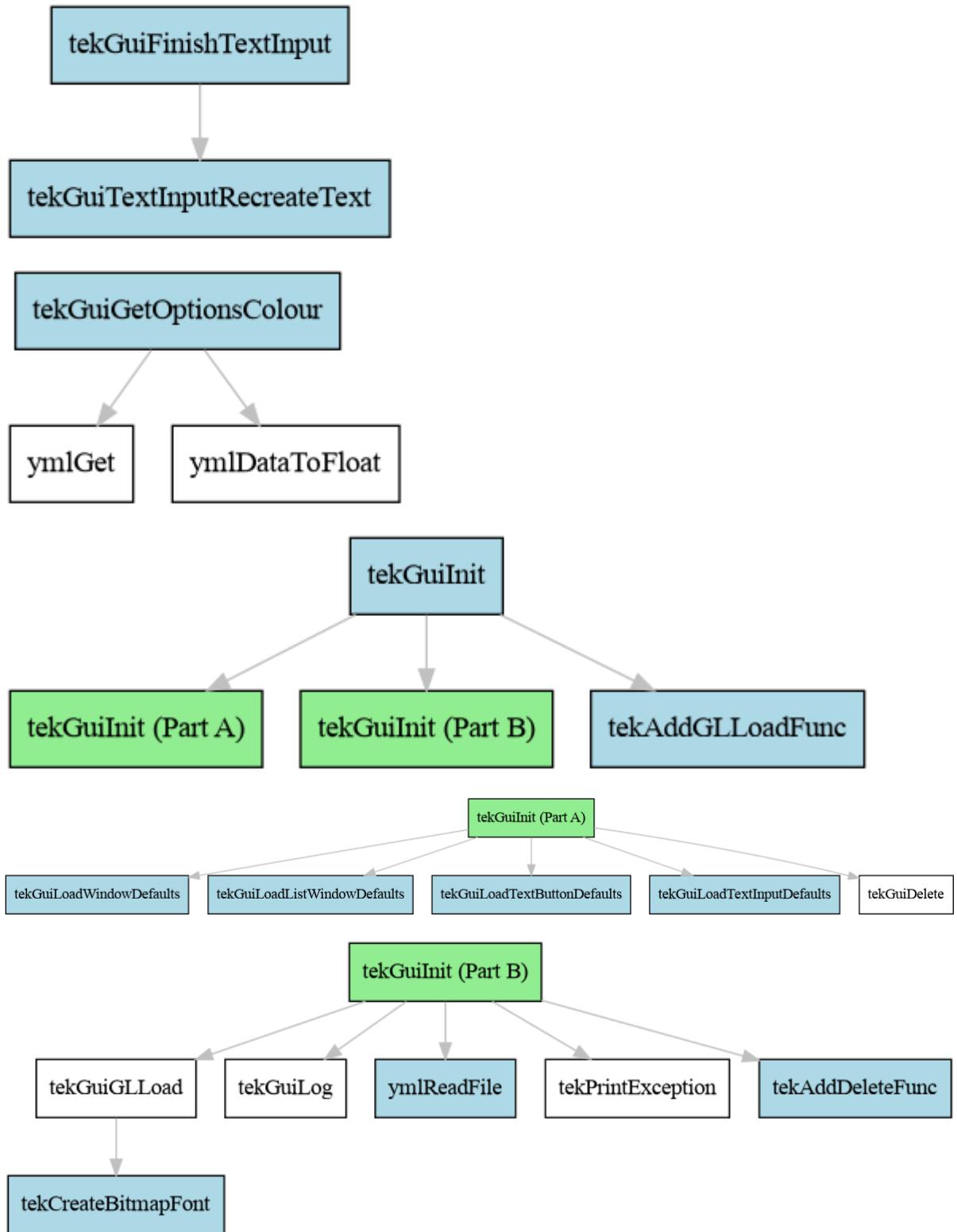


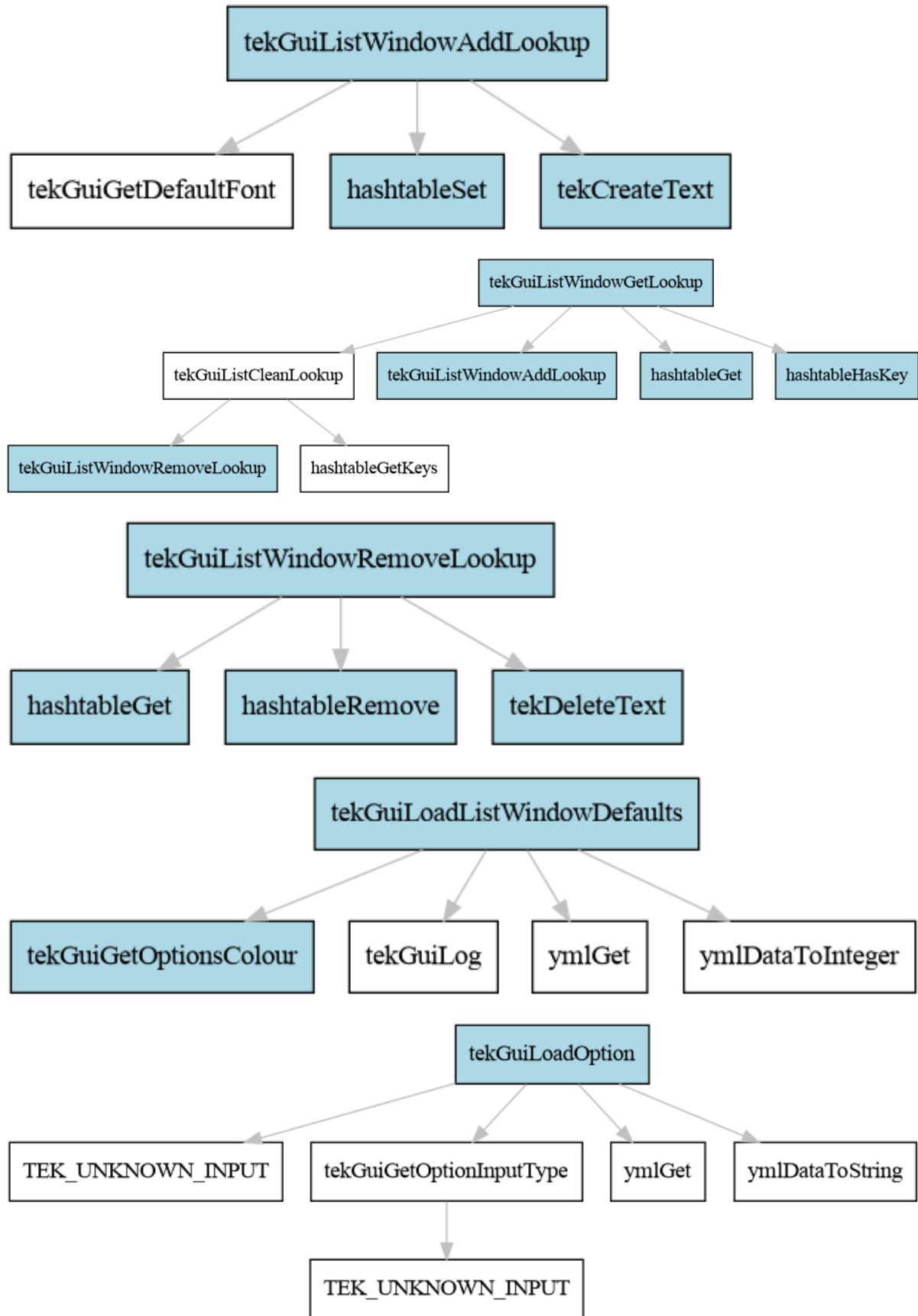






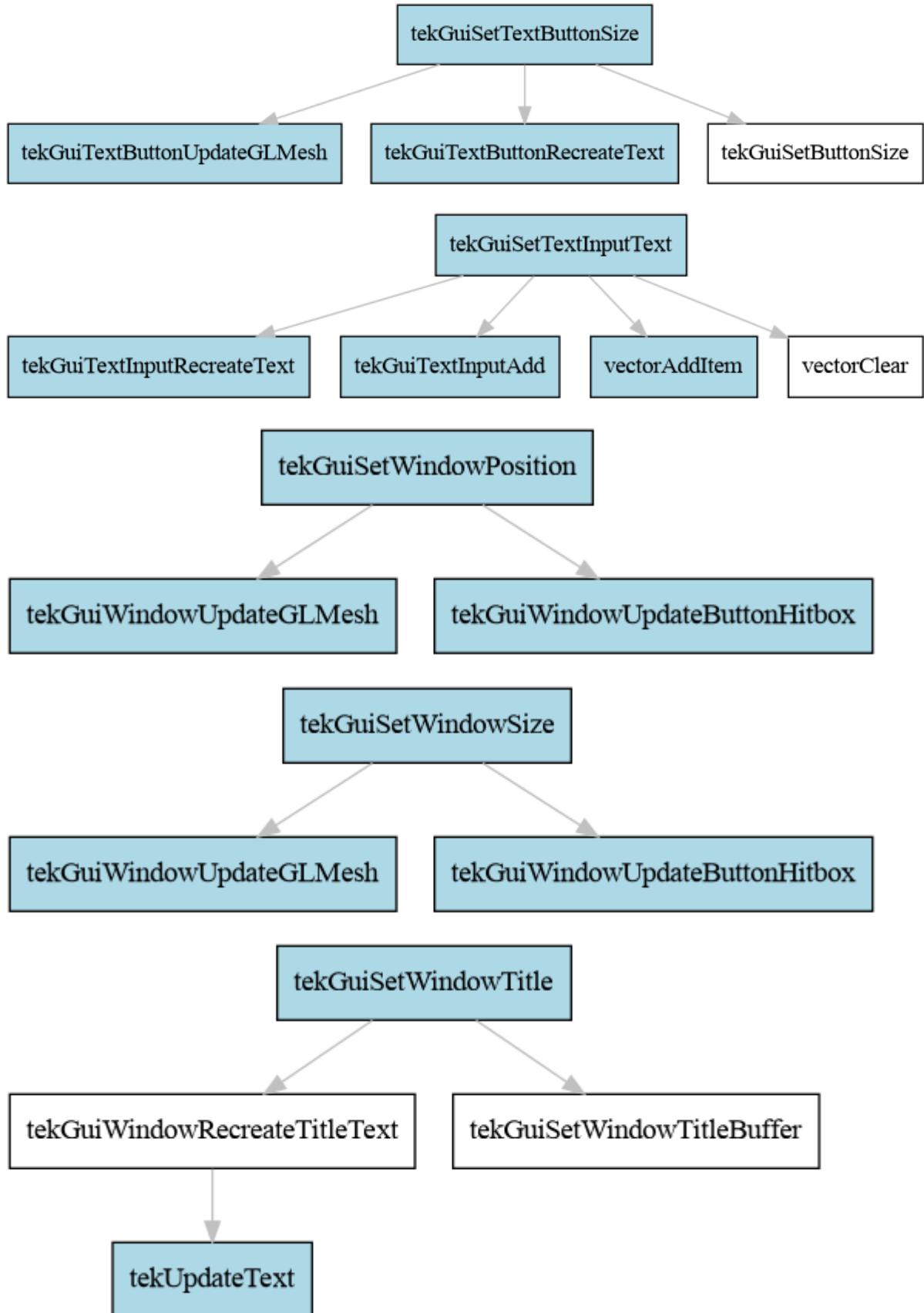


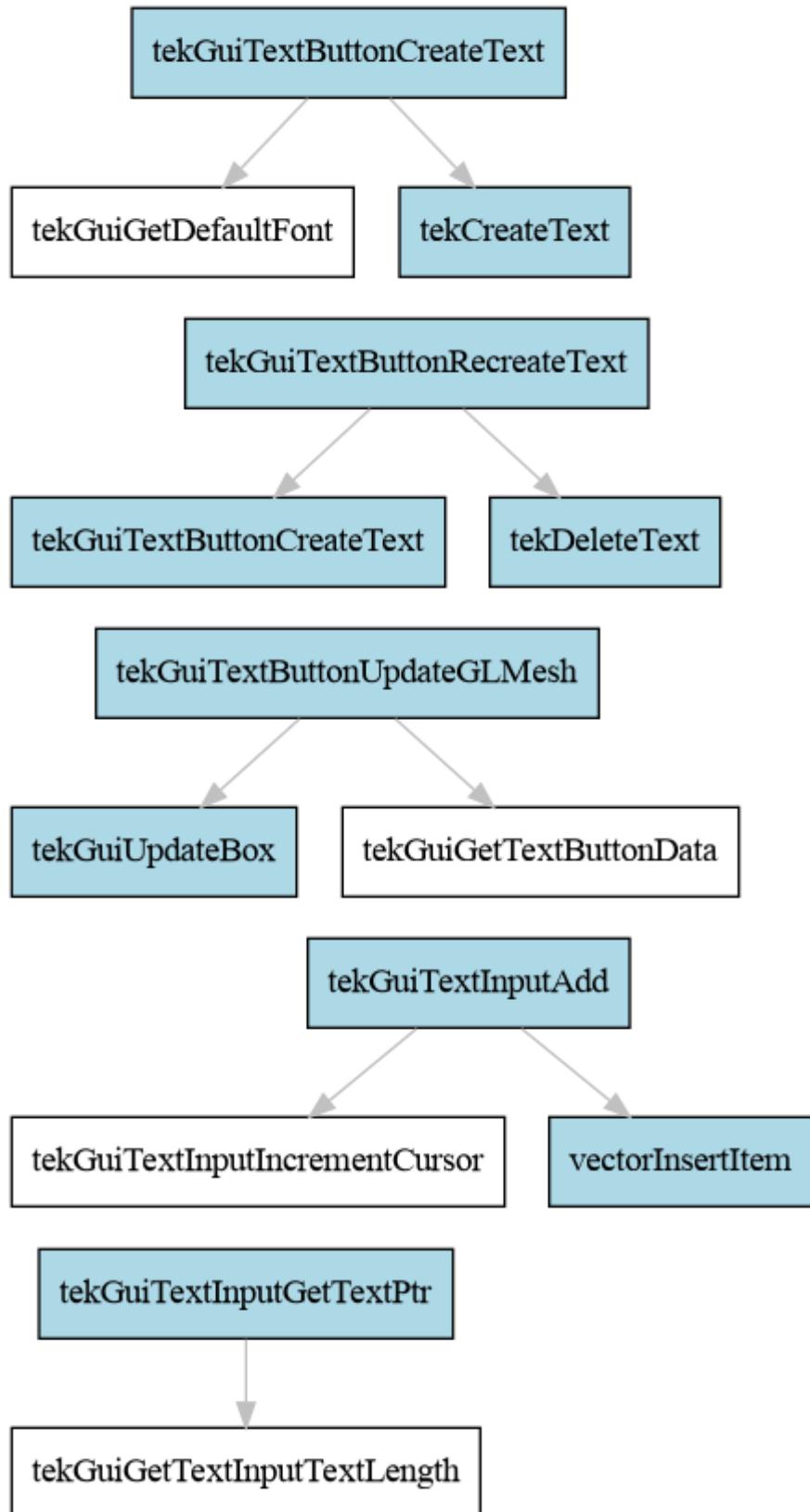


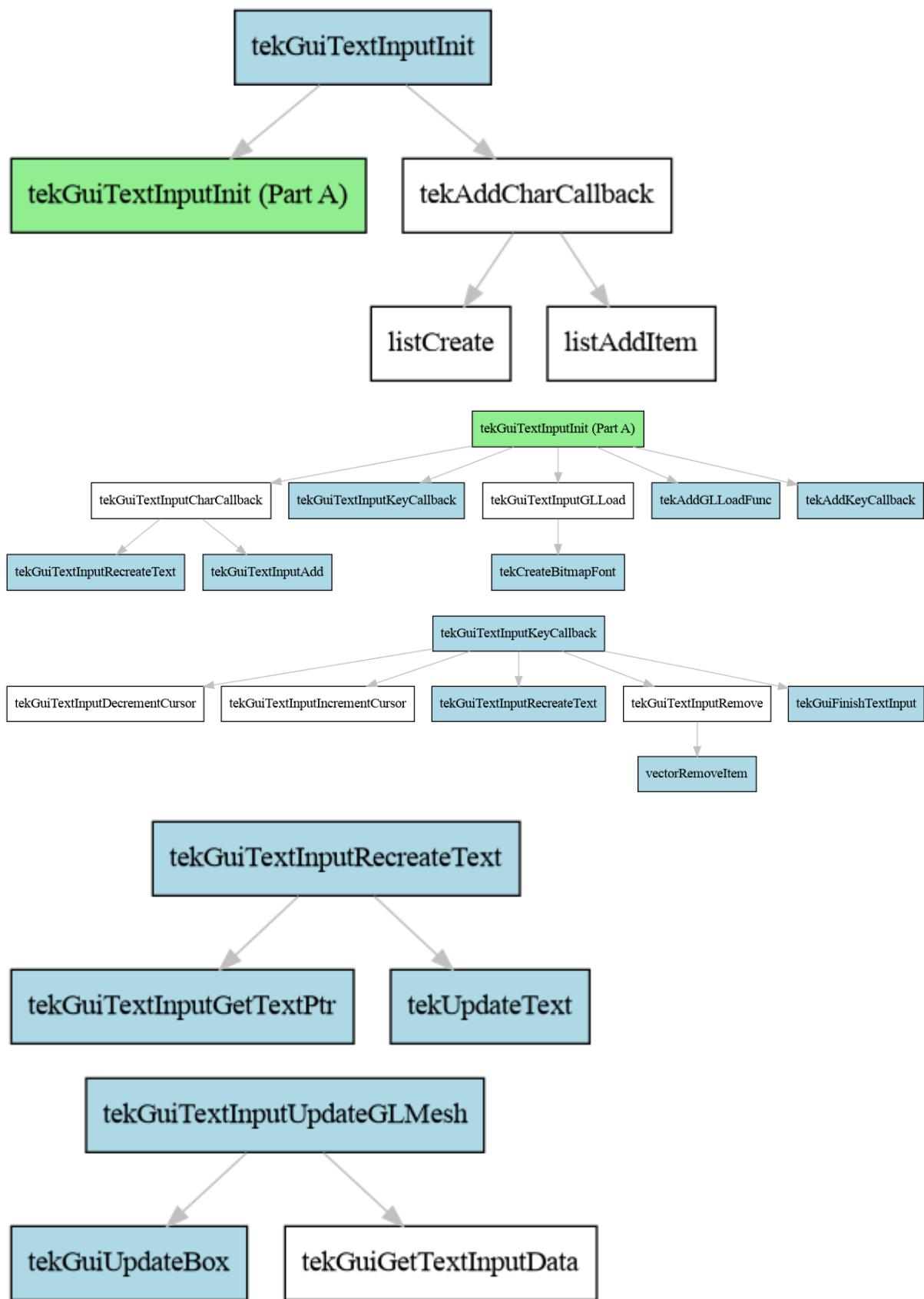


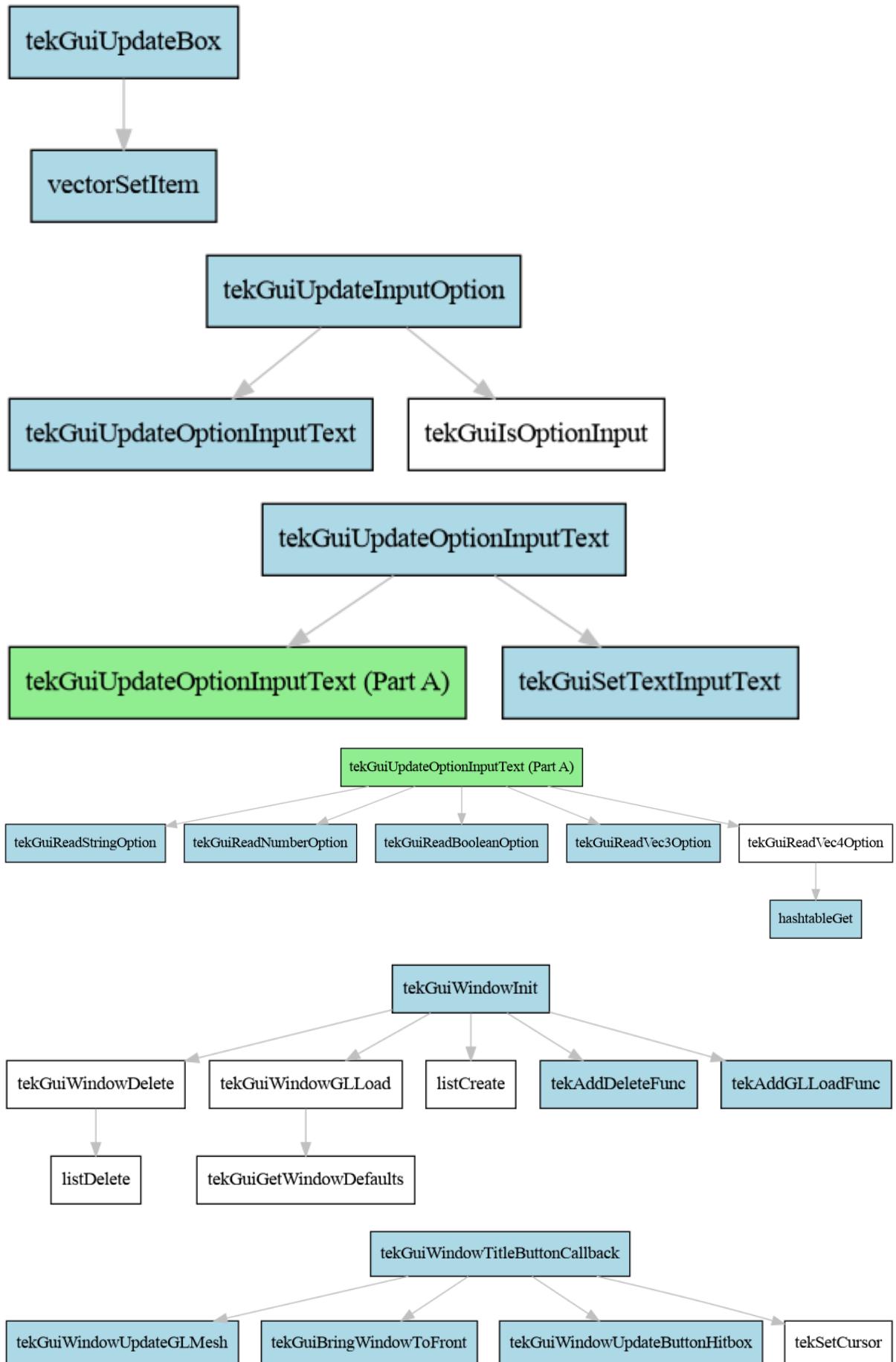


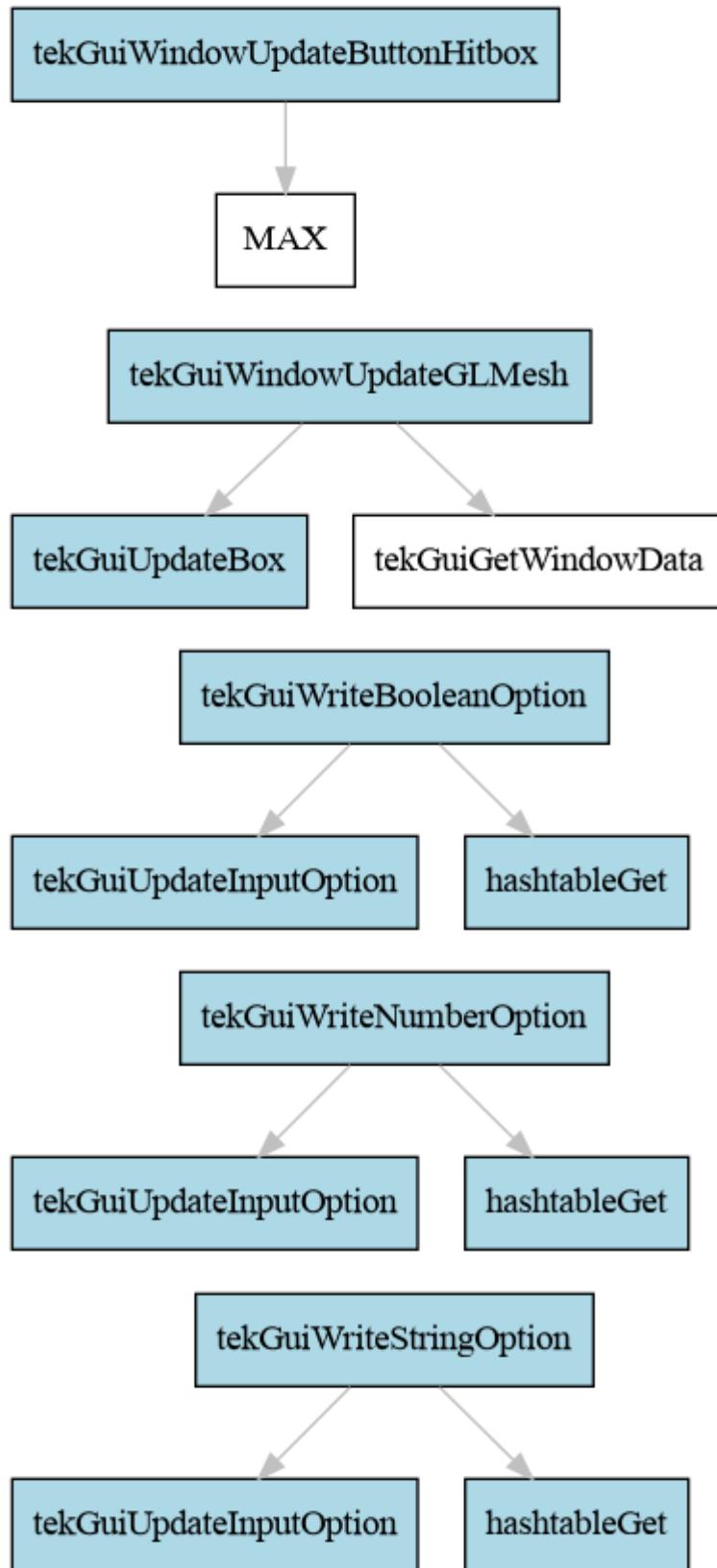


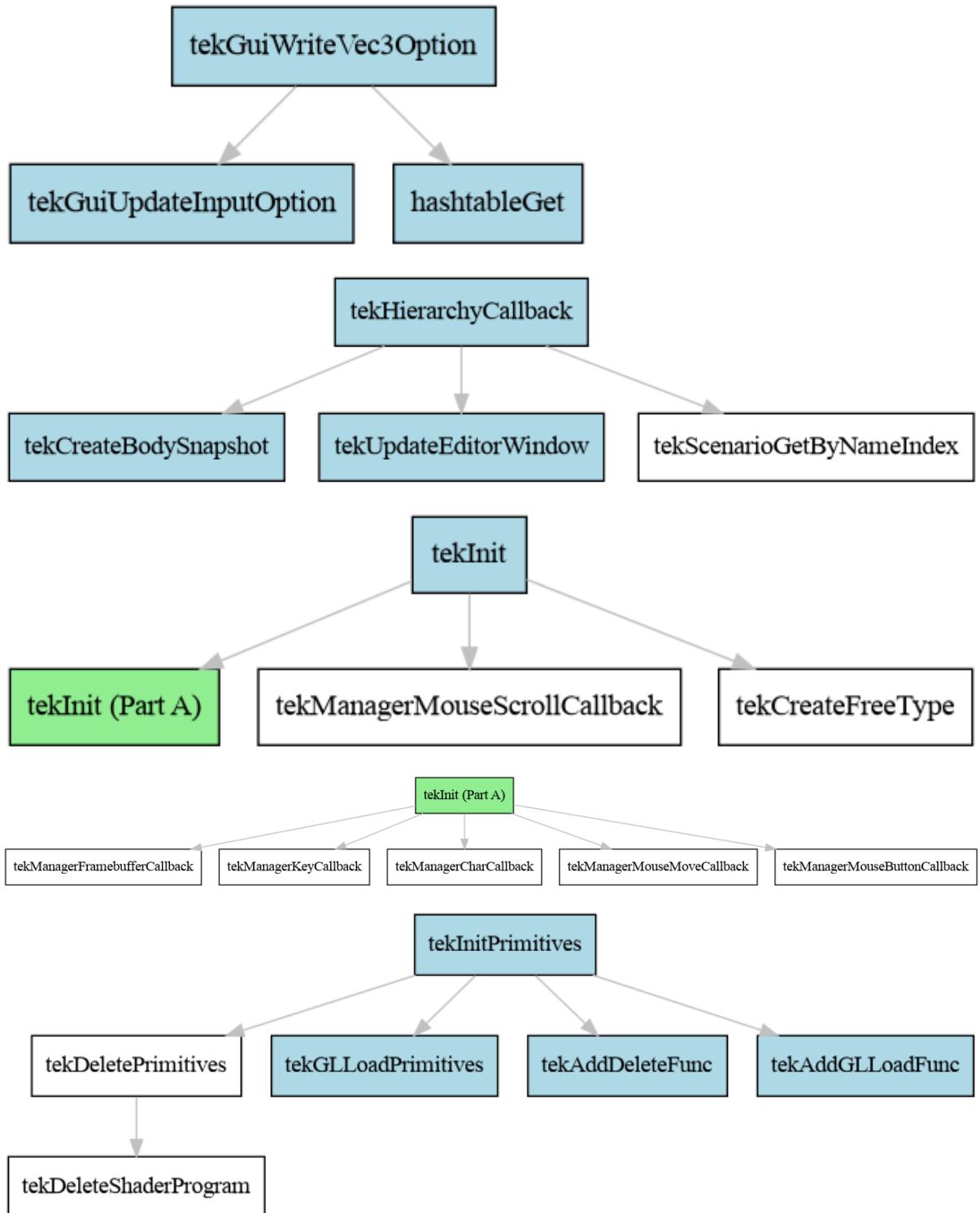


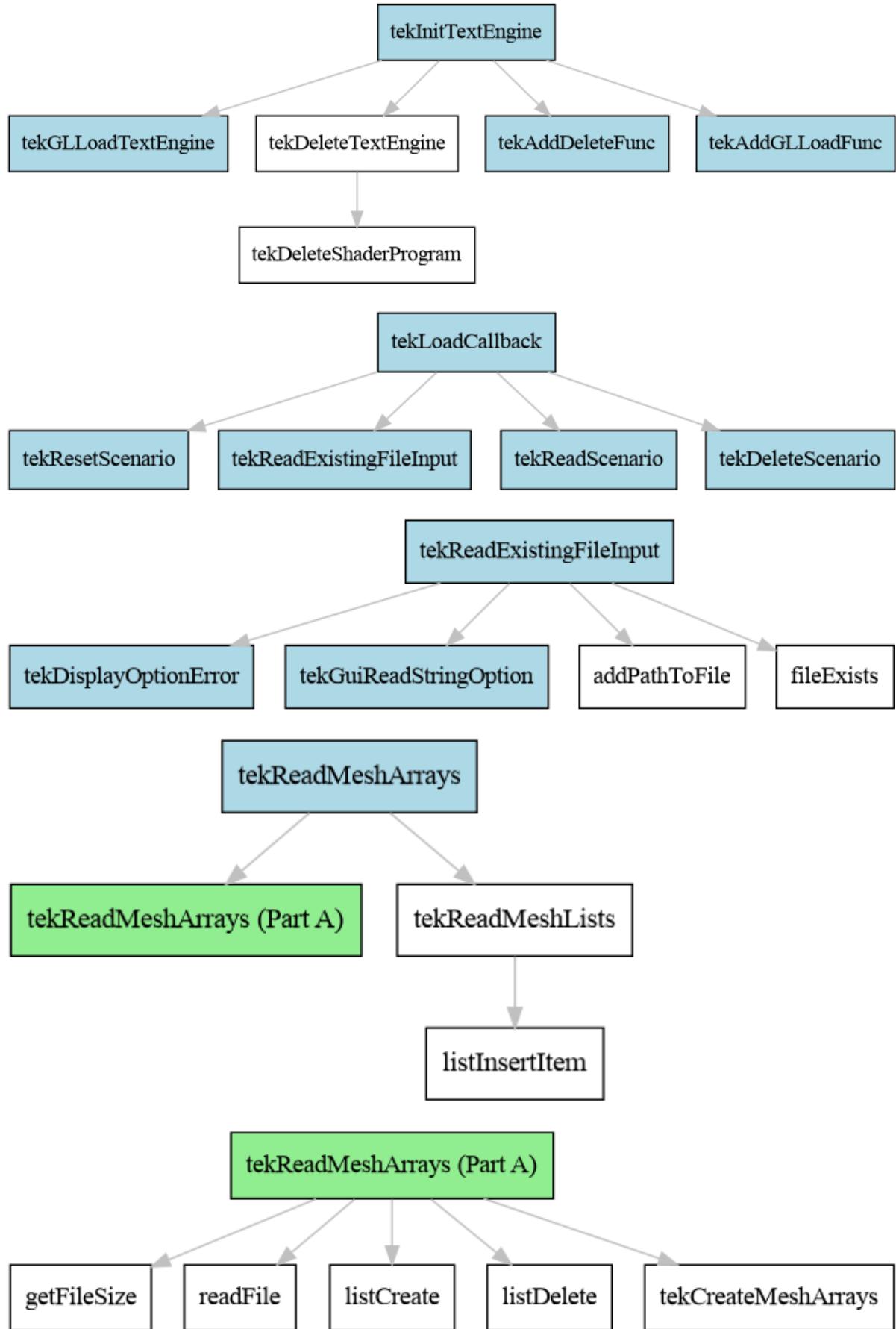


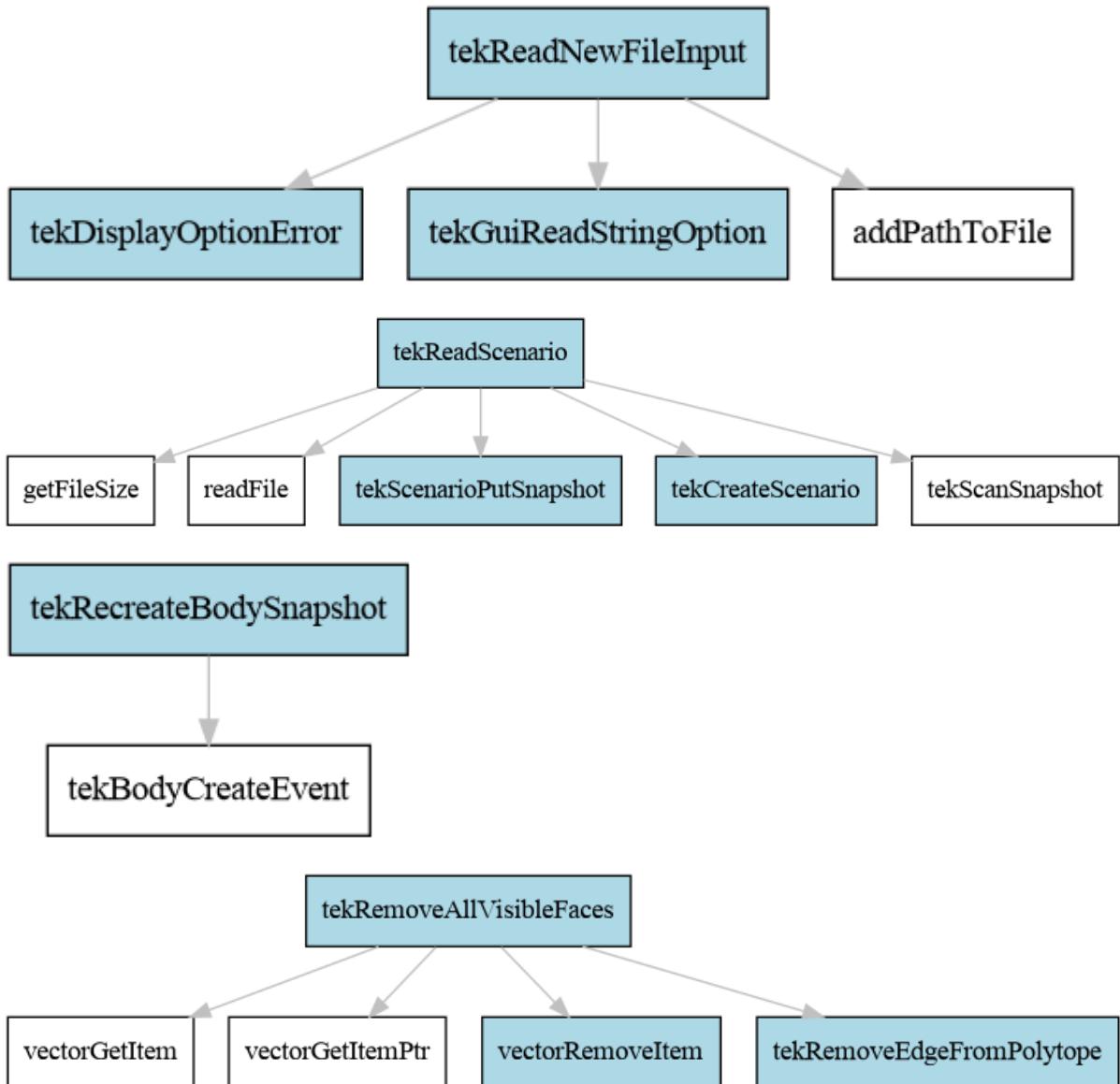


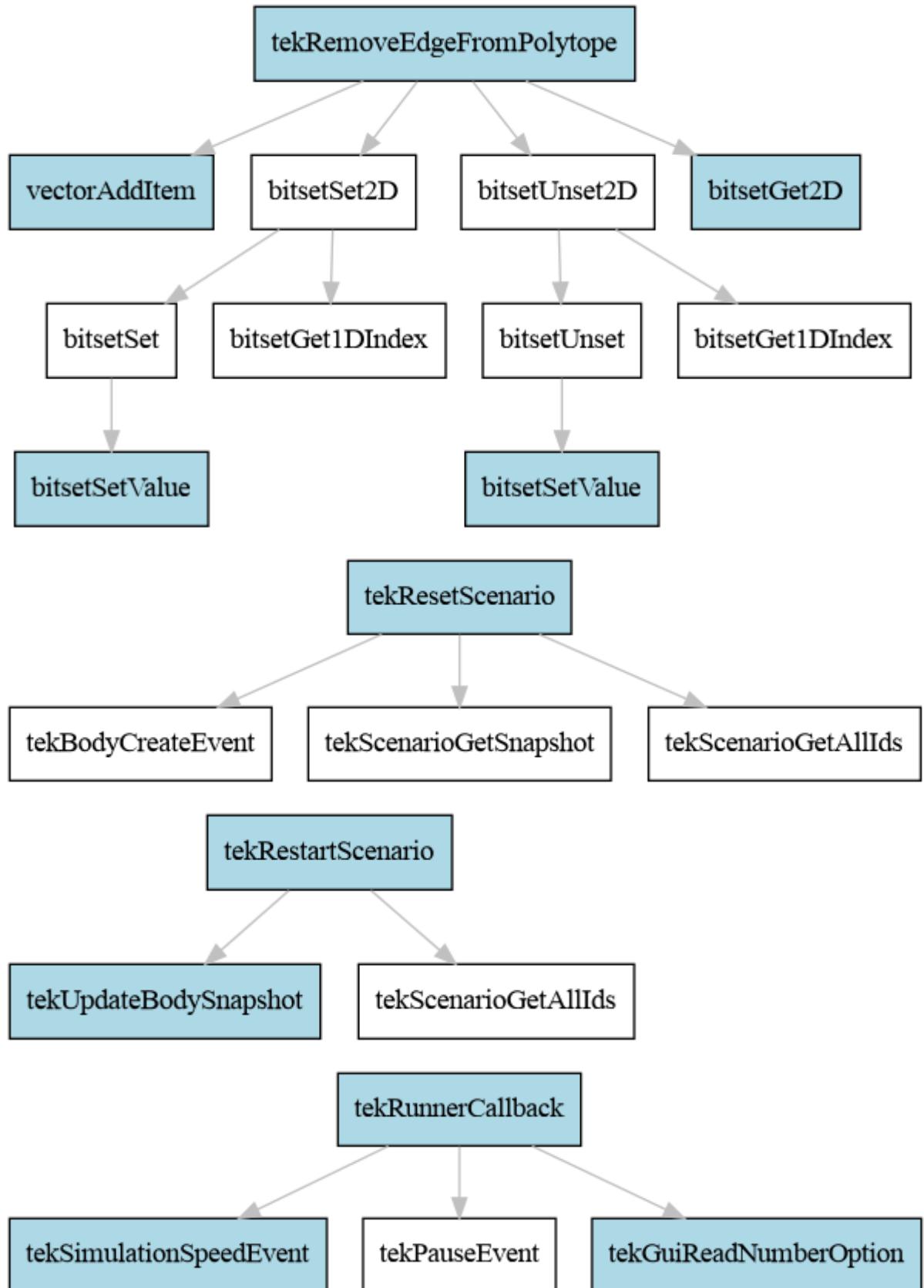


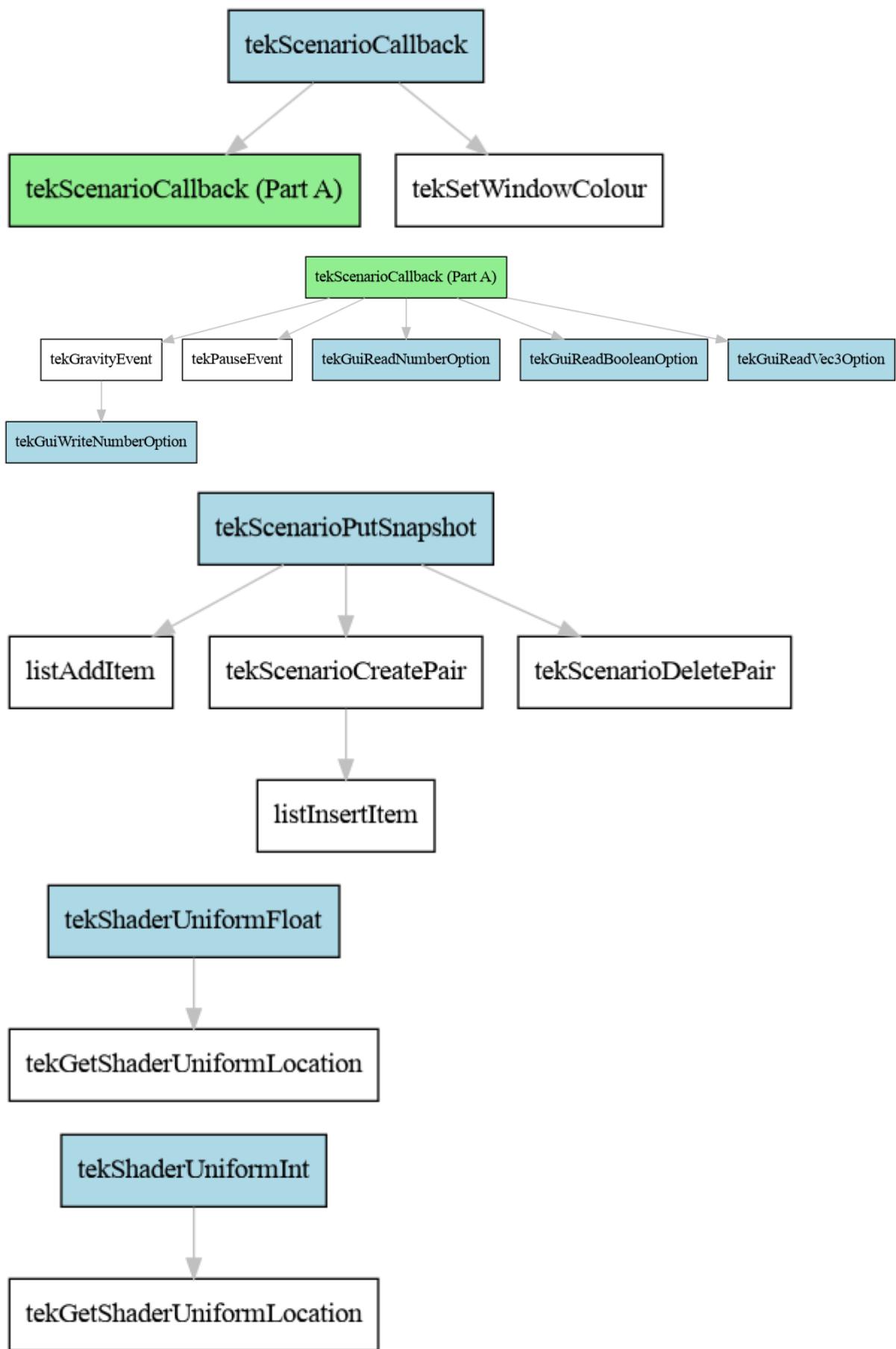


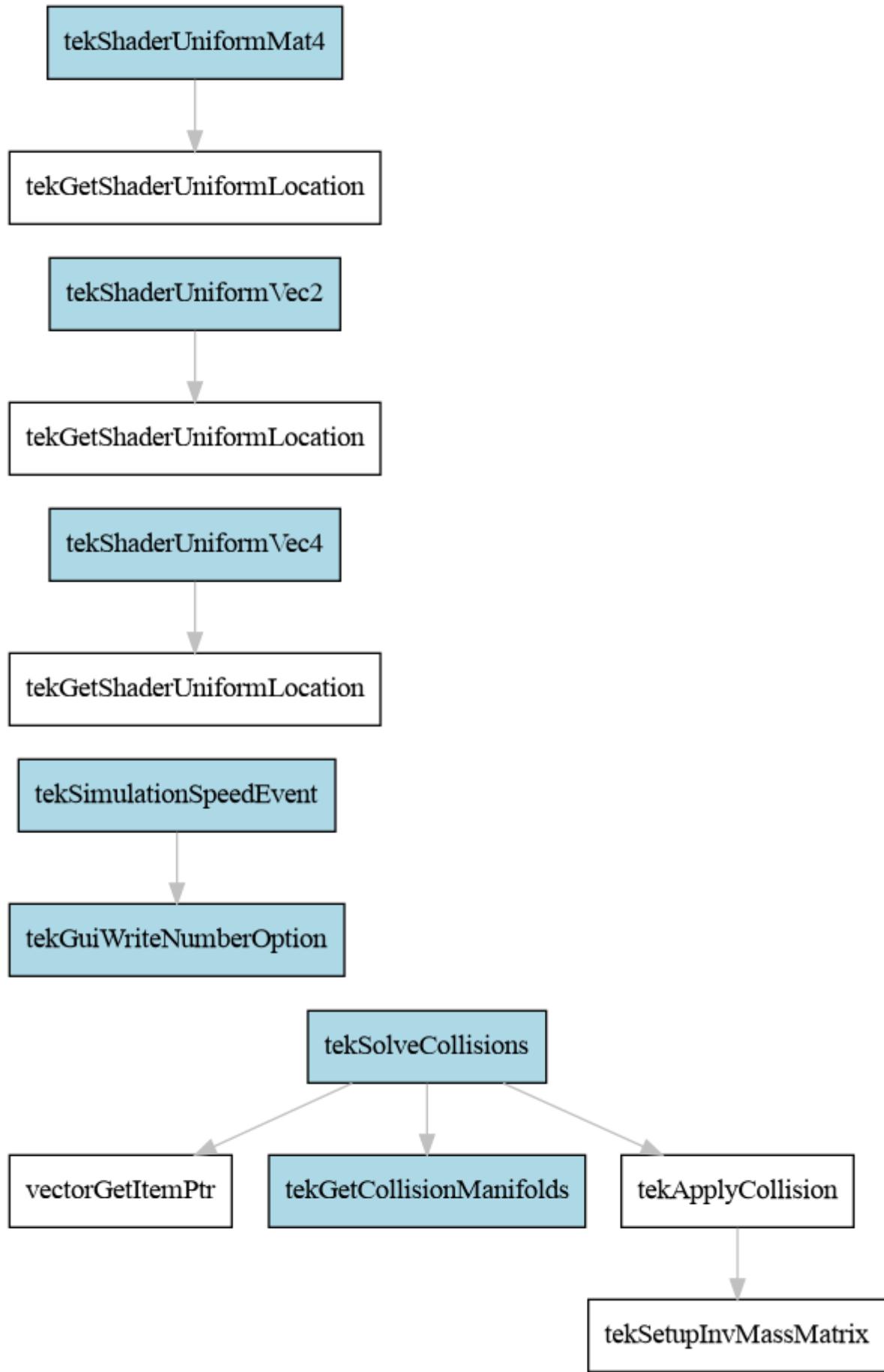




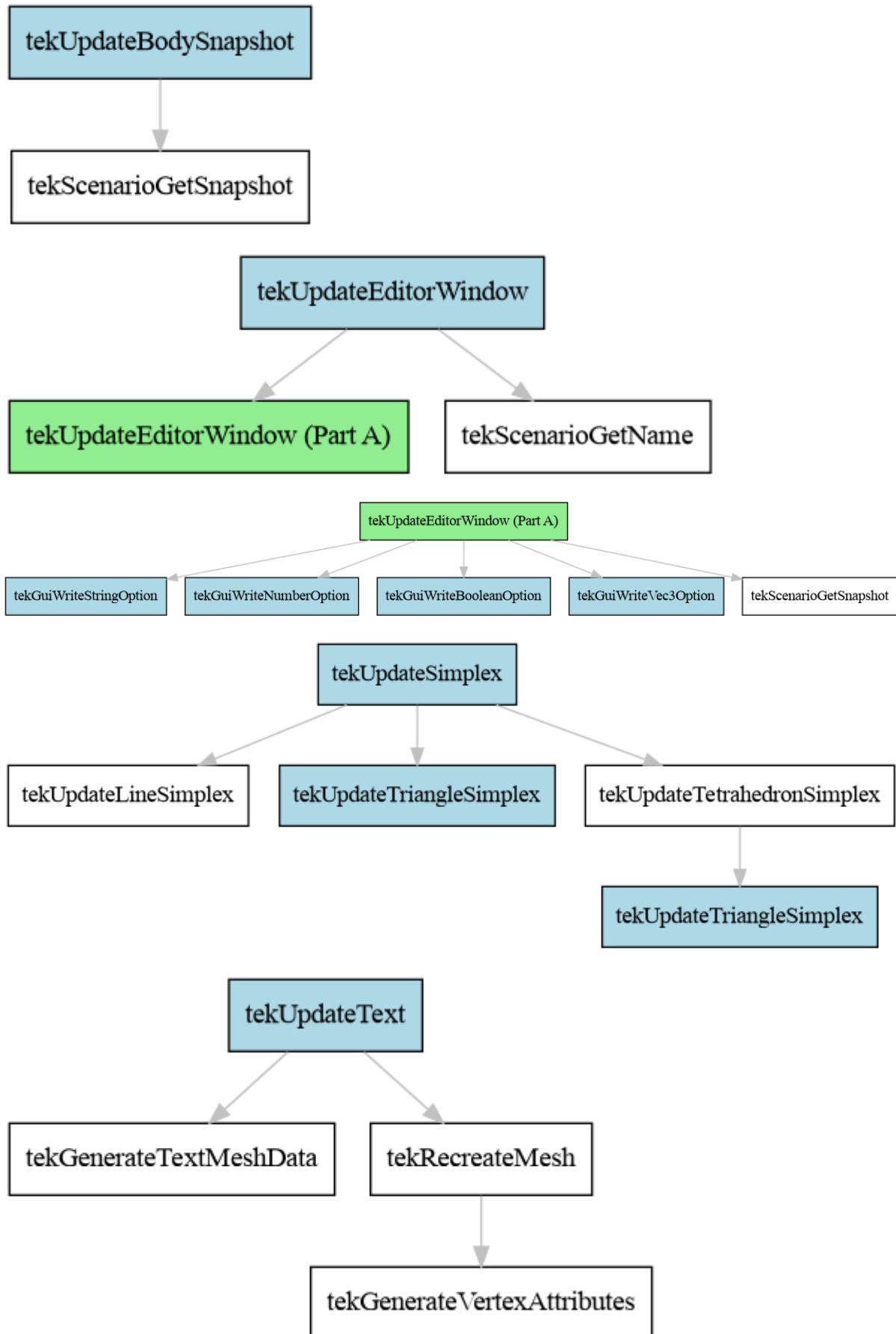


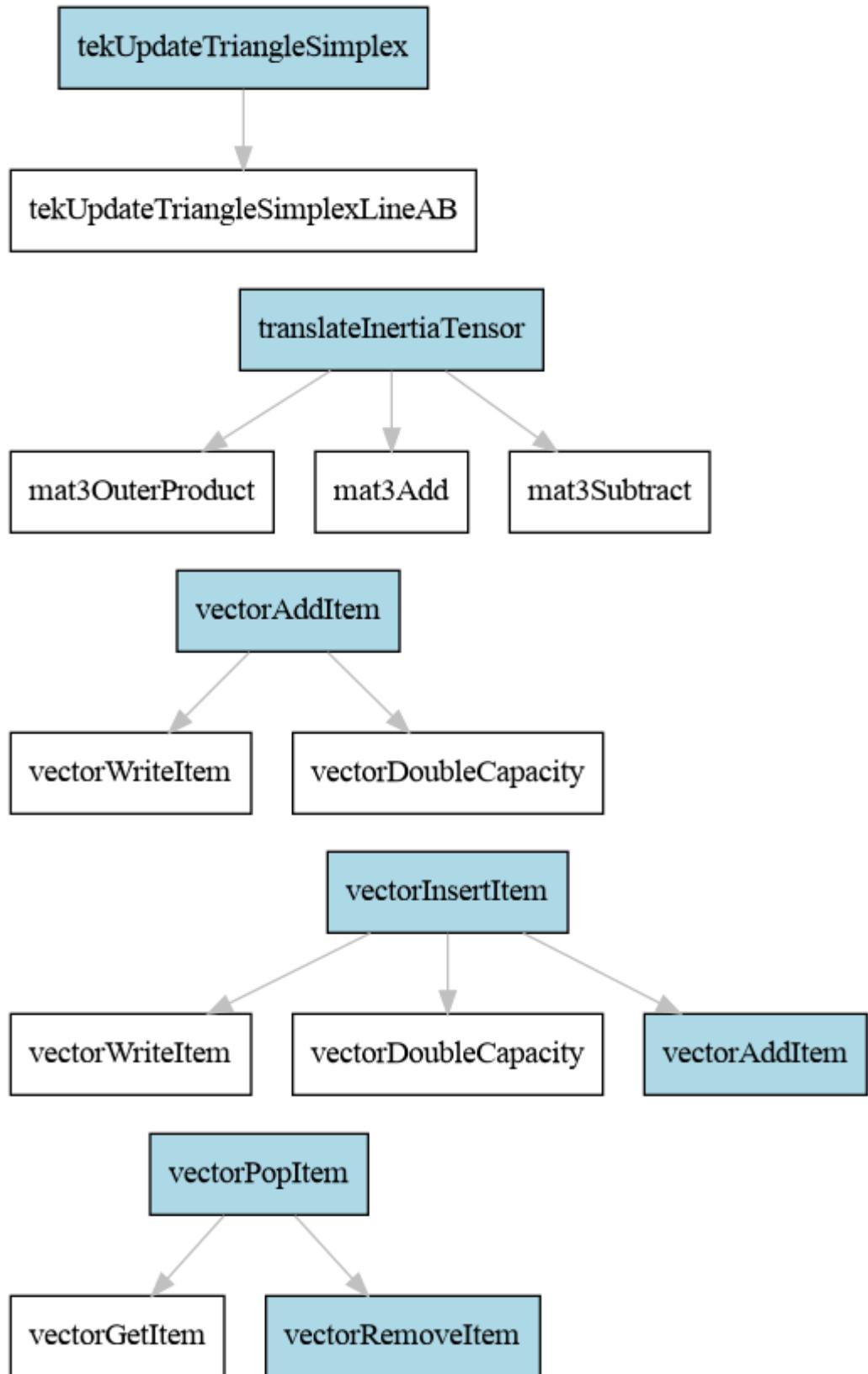


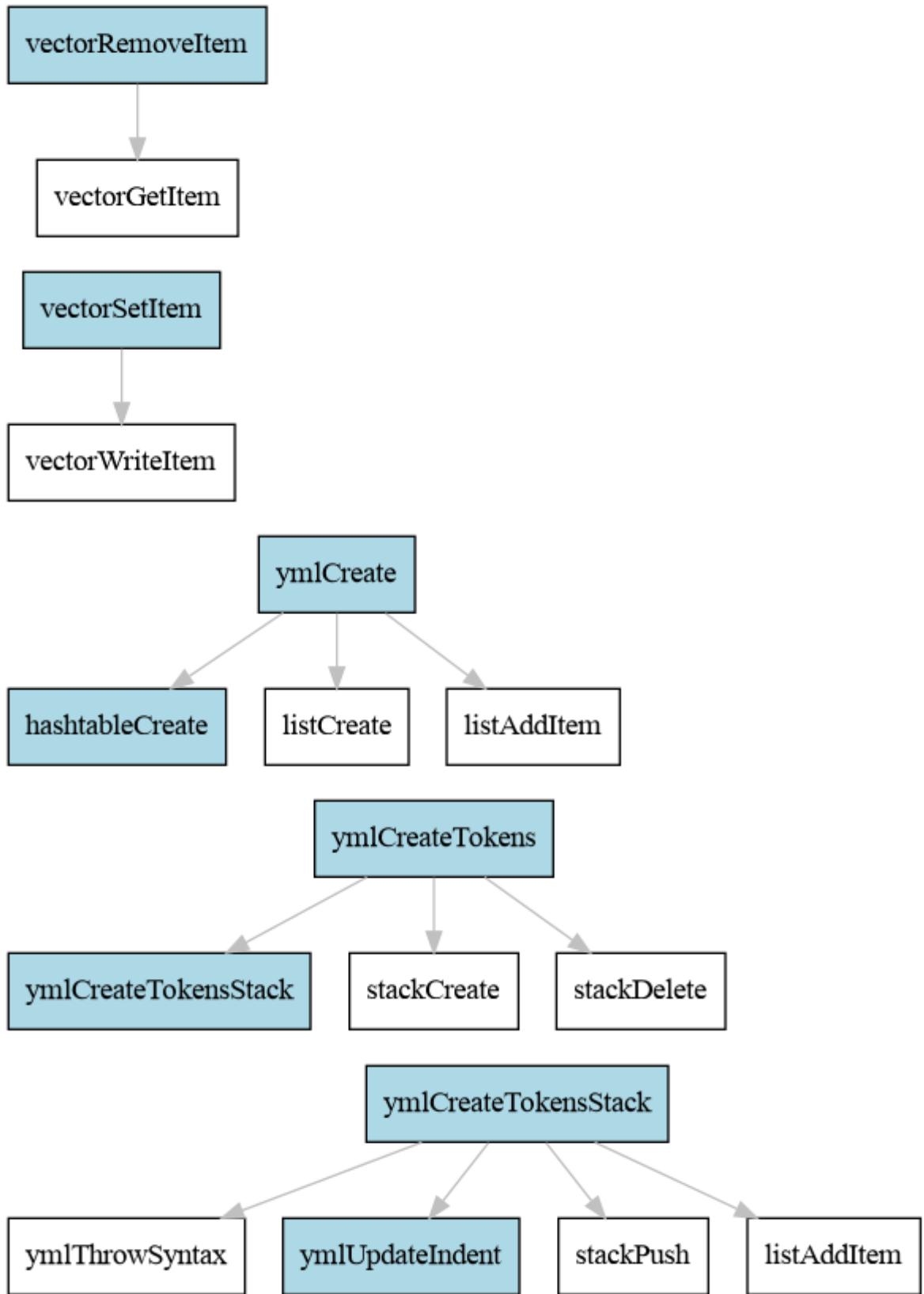


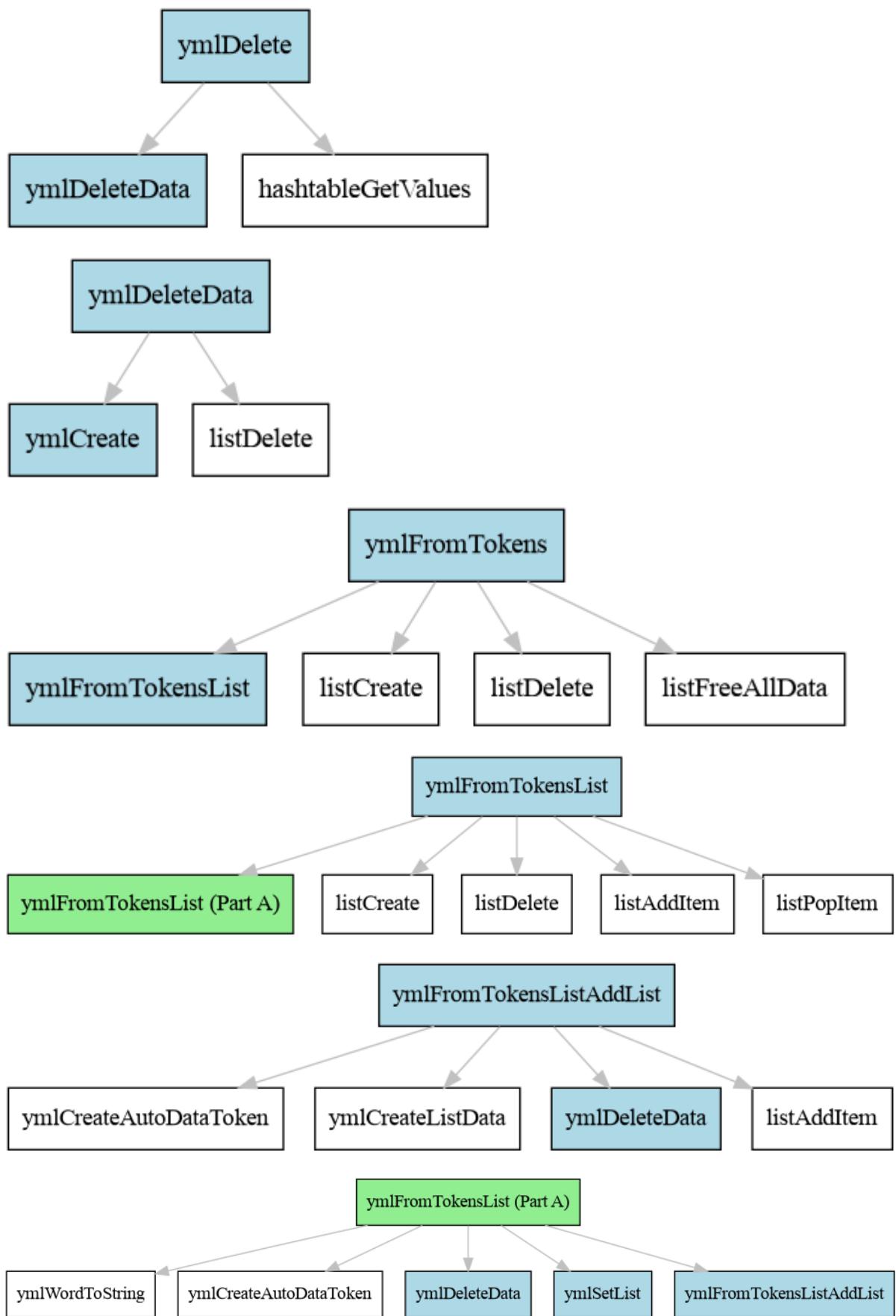


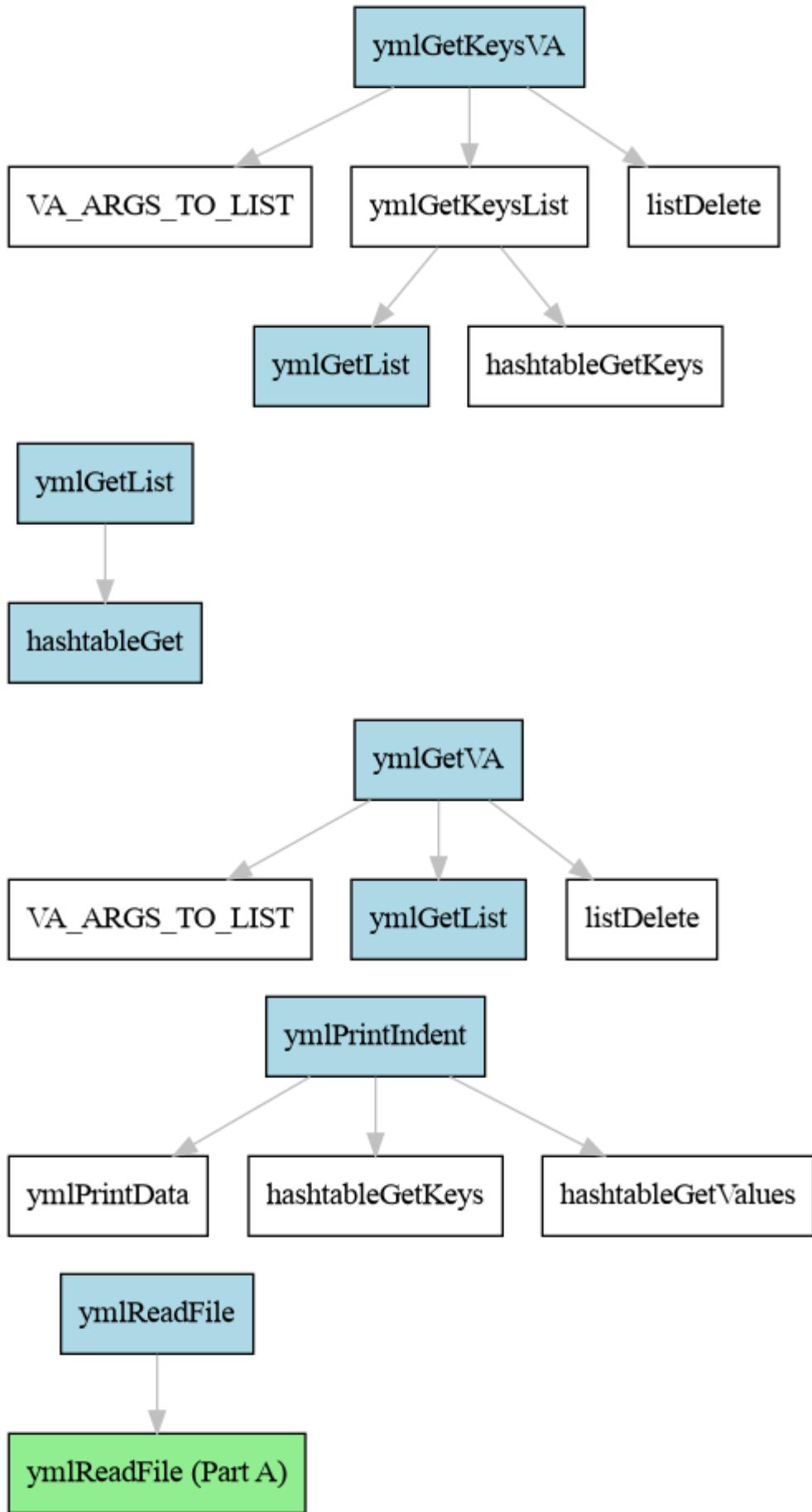


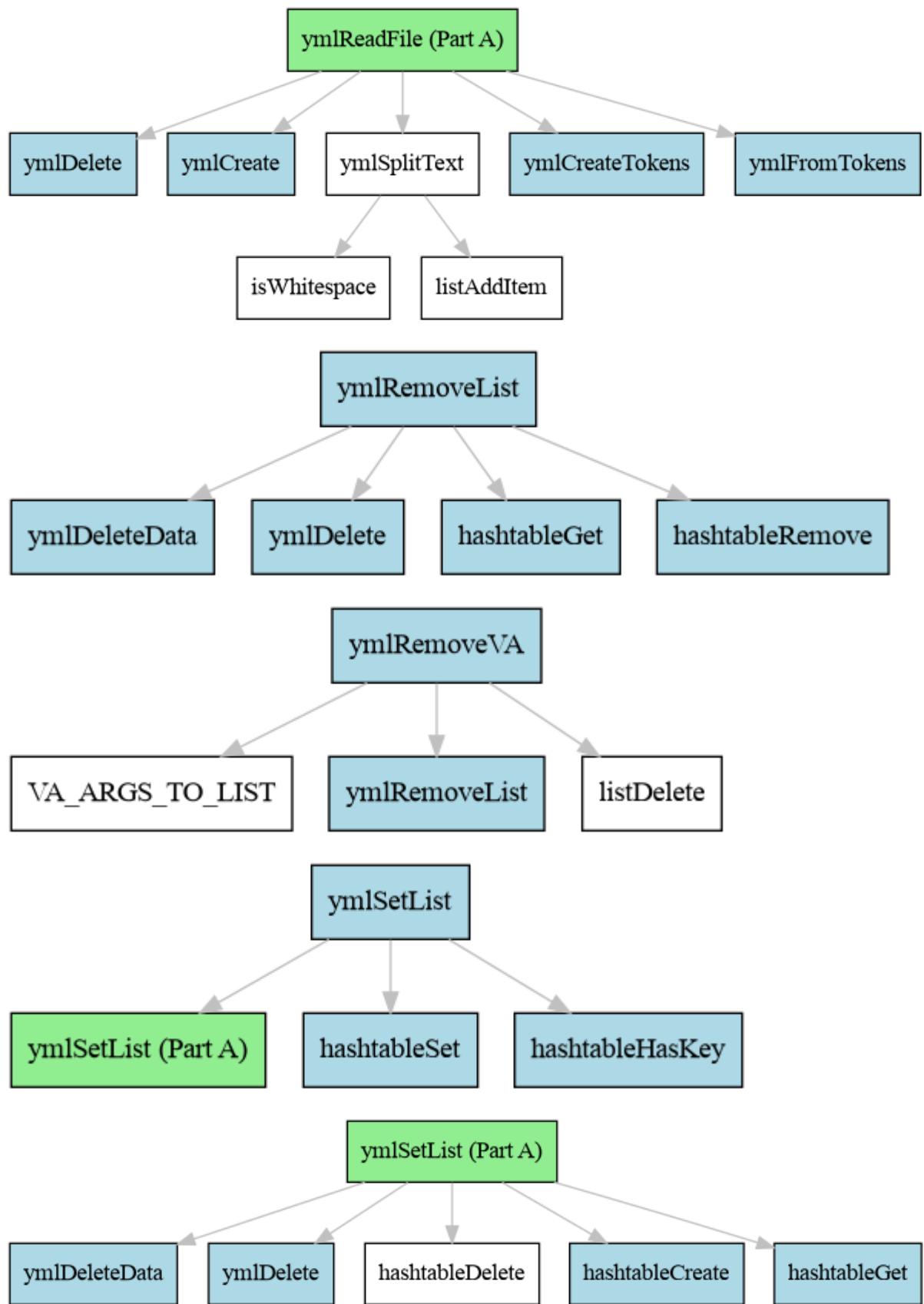


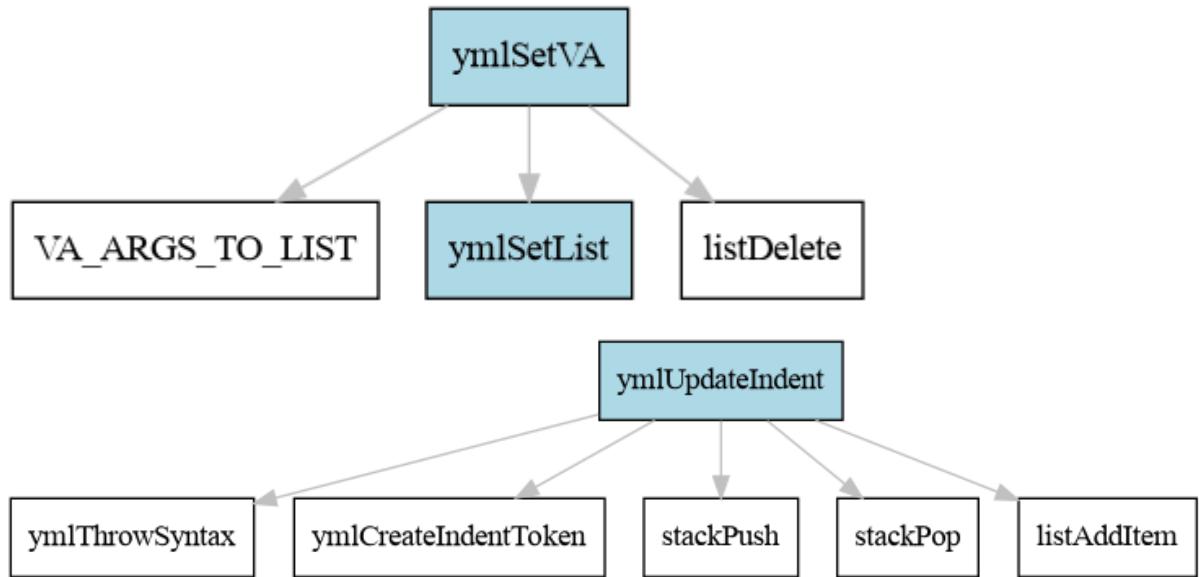












Technical solution

Code available on GitHub: <https://github.com/jcowdells/TekPhysics>

Guide to Complex Functions

The table below is a guide to all complex functions in the source code, it is intended to make the code easier to navigate and understand.

Skill + Group	Function Name	Function Description	File + Line Number
Graph / Tree Traversal, Group A	tekGetCollisionManifolds	Traverse through two collider trees simultaneously, and determine whether two colliders are colliding or not. If so, return some information about the collision (via the collision manifolds).	tekphys/collisions.c @ 1463
Linked List Maintenance, Group A	listAddItem listSetItem listDeleteItem etc.	I have created a set of functions that will handle the creation, usage and deletion of a linked list, presumably this would all come under linked list maintenance.	core/list.c @ 59, @ 98, @ 139 etc.
Hashing, Group A	hashtableHash	I created a set of functions that allow for hash tables to be used easily, this function is the one that hashes the keys to get an index for the internal array of the hash table.	core/hashtable.c @ 58
Complex User-Defined Algorithms, Group A	tekCheckTriangleCollisionC	An implementation of the GJK triangle-triangle collision detection algorithm. An iterative algorithm that works by generating an increasingly accurate Minkowski difference between two shapes, and checking if the shape contains the origin – if the origin is contained there is a collision.	tekphys/collisions.c @ 607
Complex User-Defined Algorithms,	tekApplyCollision	Part of the iterative solver that determines the changes	tekphys/collisions.c @

Group A Advanced Matrix Operations, Group A		in velocity to apply to objects in order to separate them. Uses a set of constraints with an inverse mass matrix and inverse inertia tensor matrix to calculate the changes in velocity needed.	1593
Complex User-Defined Algorithms, Group A List Operations, Group A	tekSolveCollisions	Part of the iterative solver that determines the change in velocity to separate objects. Use a nested for loop to generate all pairs of objects in the list of objects, then find which ones collide and get the collision manifold for each, and add that to another list. Then, calculate some important quantities and run the iterative solver using the calculated properties.	tekphys/collisions.c @ 1679

Source Code

main.c

```

1 #include <stdio.h>
2 #include <glad/glad.h>
3 #include <GLFW/glfw3.h>
4 #include <cglm/mat4.h>
5 #include "core/exception.h"
6 #include "tekgl/font.h"
7 #include "tekgl/text.h"
8
9 #include <cglm/cam.h>
10
11 #include "core/list.h"
12 #include "tekgl/manager.h"
13 #include "tekgui/primitives.h"
14
15 #include "tekgl/camera.h"
16
17 #include <time.h>
18 #include <math.h>
19 #include <unistd.h>
```

```

20
21 #include "core/file.h"
22 #include "core/threadqueue.h"
23 #include "core/vector.h"
24 #include "tekphys/engine.h"
25 #include "tekphys/body.h"
26 #include "tekphys/collisions.h"
27
28 #include "tekgui/tekgui.h"
29 #include "tekgui/window.h"
30 #include "tekgui/button.h"
31 #include "tekgui/list_window.h"
32 #include "tekgui/text_button.h"
33 #include "tekgui/option_window.h"
34 #include "tekphys/scenario.h"
35 #include "tests/exception_test.h"
36 #include "tests/unit_test.h"
37
38 #define WINDOW_WIDTH 1280
39 #define WINDOW_HEIGHT 720
40
41 #define START_BUTTON_WIDTH 100
42 #define START_BUTTON_HEIGHT 50
43
44 #define LOGO_WIDTH 300.0f
45 #define LOGO_HEIGHT 50.0f
46
47 // indices of possible options
48 #define SAVE_OPTION 0
49 #define LOAD_OPTION 1
50 #define RUN_OPTION 2
51 #define QUIT_OPTION 3
52 #define NUM_OPTIONS 4
53
54 #define MOUSE_SENSITIVITY 0.5f
55 #define MOVE_SPEED 2.0f
56
57 #define DEFAULT_RATE 100.0
58 #define DEFAULT_SPEED 1.0
59
60 struct TekScenarioOptions {
61     float gravity;
62     double rate;

```

```

63     double speed;
64     flag pause;
65     vec3 sky_colour;
66 };
67
68 struct TekGuiComponents {
69     TekGuiTextButton start_button;
70     TekGuiImage logo;
71     TekText version_text;
72     TekText splash_text;
73     TekGuiListWindow hierarchy_window;
74     TekGuiOptionWindow editor_window;
75     TekGuiListWindow action_window;
76     TekGuiOptionWindow scenario_window;
77     TekGuiOptionWindow save_window;
78     TekGuiOptionWindow load_window;
79     TekGuiOptionWindow runner_window;
80     TekGuiWindow inspect_window;
81     TekText inspect_text;
82 };
83
84 static flag mode = MODE_MAIN_MENU;
85 static flag next_mode = -1;
86 static int hierarchy_index = -1, inspect_index = -1;
87
88 ThreadQueue event_queue = {};
89
90 double mouse_x = 0.0, mouse_y = 0.0;
91 float mouse_dx = 0.0f, mouse_dy = 0.0f;
92 flag mouse_moved = 0, mouse_right = 0;
93 flag w_pressed = 0, a_pressed = 0, s_pressed = 0, d_pressed =
0, up_pressed = 0, down_pressed = 0;
94 TekScenario active_scenario = {};
95
96 /**
97  * Push an inspect event to the event queue. Changes which
body is shown to the display.
98  * @param inspect_id The ID of the body to inspect.
99  * @throws MEMORY_EXCEPTION if malloc() fails.
100 */
101 static exception tekPushInspectEvent(const uint inspect_id) {
102     // create event
103     TekEvent event = {};

```

```

104     event.type = INSPECT_EVENT;
105     event.data.body.id = inspect_id;
106
107
108     // push event to event queue
109     tekChainThrow(pushEvent(&event_queue, event));
110     return SUCCESS;
111 }
112
113 /**
114  * The main key callback of the program. Listens for the W, A,
S and D keys to allow for camera to move.
115  * @param key The keycode of the key that was pressed.
116  * @param scancode The scancode of the key.
117  * @param action The action of the key - pressed, released,
repeated.
118  * @param mods Any modifiers on the key e.g. shift.
119 */
120 void tekMainKeyCallback(const int key, const int scancode,
const int action, const int mods) {
121     // listen for key presses, and update if the key is being
pressed
122     // WASD for movement controls
123     if (key == GLFW_KEY_W) {
124         if (action == GLFW_RELEASE) w_pressed = 0;
125         else w_pressed = 1;
126     }
127     if (key == GLFW_KEY_A) {
128         if (action == GLFW_RELEASE) a_pressed = 0;
129         else a_pressed = 1;
130     }
131     if (key == GLFW_KEY_S) {
132         if (action == GLFW_RELEASE) s_pressed = 0;
133         else s_pressed = 1;
134     }
135     if (key == GLFW_KEY_D) {
136         if (action == GLFW_RELEASE) d_pressed = 0;
137         else d_pressed = 1;
138     }
139
140     // arrows change inspect index and send inspect event
141     // up arrow decreases the inspect index (goes up the list)
142     if (key == GLFW_KEY_UP && action == GLFW_RELEASE) {

```

```

143         if (inspect_index > -1) {
144             inspect_index--;
145             if (inspect_index == -1) tekPushInspectEvent(0);
146             else tekPushInspectEvent((uint)inspect_index);
147         }
148     }
149
150     // down arrow increases the inspect index (goes down the
list)
151     if (key == GLFW_KEY_DOWN && action == GLFW_RELEASE) {
152         if (inspect_index + 2 < active_scenario.names.length)
{
153             inspect_index++;
154             tekPushInspectEvent((uint)inspect_index);
155         }
156     }
157
158 }
159
160 /**
161 * The main mouse position callback of the program. Used to
update the camera rotation.
162 * @param x The new x position of the mouse relative to the
window.
163 * @param y The new y position of the mouse relative to the
window.
164 */
165 void tekMainMousePosCallback(const double x, const double y) {
166     // if the mouse has already been moved since last checked,
then dont update again
167     if (mouse_moved) return;
168
169     // update positions and deltas
170     mouse_dx = (float)(x - mouse_x);
171     mouse_dy = (float)(y - mouse_y);
172     mouse_x = x;
173     mouse_y = y;
174
175     // mark that there has been a mouse update
176     mouse_moved = 1;
177 }
178
179 /**

```

```
180     * The main mouse button callback of the program. Used to
181     * track if the right mouse button is pressed, which would determine if
182     * the camera should rotate or not.
183
184     */
185     void tekMainMouseButtonCallback(const int button, const int
186 action, const int mods) {
187         // keep track of whether right clicking or not.
188         // used for camera control
189         if (button == GLFW_MOUSE_BUTTON_RIGHT) {
190             if (action == GLFW_PRESS) {
191                 mouse_right = 1;
192             } else if (action == GLFW_RELEASE) {
193                 mouse_right = 0;
194             }
195         }
196
197     /**
198     * Push a body create event to the event queue.
199     * @param snapshot A pointer to a body snapshot.
200     * @param snapshot_id The id of the body to update using this
snapshot.
201     * @note The body snapshot is copied into the event.
202     * @throws MEMORY_EXCEPTION if the event could not be added to
the queue.
203     */
204     static exception tekBodyCreateEvent(const TekBodySnapshot*
snapshot, const int snapshot_id) {
205         // create event
206         TekEvent event = {};
207         event.type = BODY_CREATE_EVENT;
208         memcpy(&event.data.body.snapshot, snapshot,
209 sizeof(TekBodySnapshot));
210         event.data.body.id = snapshot_id;
211
212         // push event to event queue
213         tekChainThrow(pushEvent(&event_queue, event));
214         return SUCCESS;
215     }
```

```

216 /**
217  * Create a body snapshot filled with default values and add
218 it to a scenario. Also chooses a new ID for the created body.
219 * @param scenario The scenario for which to add the new
220 snapshot.
221 * @param snapshot_id A pointer to where the newly created ID
222 is to be written to.
223 * @throws MEMORY_EXCEPTION if malloc() fails.
224 */
225 static exception tekCreateBodySnapshot(TekScenario* scenario,
226 int* snapshot_id) {
227     // make a dummy snapshot
228     TekBodySnapshot dummy = {};
229
230     // default values
231     dummy.mass = 1.0f;
232     dummy.friction = 0.5f;
233     dummy.restitution = 0.5f;
234     glm_vec3_zero(dummy.position);
235     glm_vec3_zero(dummy.rotation);
236     glm_vec3_zero(dummy.velocity);
237     glm_vec3_zero(dummy.angular_velocity);
238     dummy.immovable = 0;
239
240     // default strings - need to be copied
241     const char* default_model = "../res/rad1.tmsh";
242     const uint len_default_model = strlen(default_model) + 1;
243     const char* default_material = "../res/material.tmat";
244     const uint len_default_material = strlen(default_material)
+ 1;
245
246     // allocate buffer for strings
247     dummy.model = malloc(len_default_model * sizeof(char));
248     if (!dummy.model)
249         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for model filepath.");
250     dummy.material = malloc(len_default_material *
sizeof(char));
251     if (!dummy.material) {
252         free(dummy.model);
253         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for material filepath");
254     }

```

```

251
252     // copy into buffers
253     memcpy(dummy.model, default_model, len_default_model);
254     memcpy(dummy.material, default_material,
255         len_default_material);
256     // find next id, could include gaps where bodies were
257     // deleted.
258     uint id;
259     tekChainThrowThen(tekScenarioGetNextId(scenario, &id), {
260         free(dummy.model);
261         free(dummy.material);
262     });
263     tekChainThrowThen(tekScenarioPutSnapshot(scenario, &dummy,
264 id, "New Object"), {
265         free(dummy.model);
266         free(dummy.material);
267     });
268     *snapshot_id = (int)id;
269     // push event to event queue
270     tekChainThrowThen(tekBodyCreateEvent(&dummy, id), {
271         free(dummy.model);
272         free(dummy.material);
273     });
274     return SUCCESS;
275 }
276
277 /**
278  * Push a body update event to the event queue. Uses the body
279  * snapshot data stored at the specified ID in the scenario.
280  * @param scenario The scenario where the body is stored.
281  * @param snapshot_id The ID of this body.
282  * @throws MEMORY_EXCEPTION if malloc() fails.
283  * @throws FAILURE if the body snapshot was not found in the
284  * scenario.
285  */
286 static exception tekUpdateBodySnapshot(const TekScenario*
scenario, const int snapshot_id) {
287     // in case no body selected
288     if (snapshot_id < 0)
289         return SUCCESS;

```

```

288
289     // get snapshot with that id
290     TekBodySnapshot* body_snapshot;
291     tekChainThrow(tekScenarioGetSnapshot(scenario,
snapshot_id, &body_snapshot));
292
293     // create event
294     TekEvent event = {};
295     event.type = BODY_UPDATE_EVENT;
296     memcpy(&event.data.body.snapshot, body_snapshot,
sizeof(TekBodySnapshot));
297     event.data.body.id = (uint)snapshot_id;
298
299     // push event to event queue
300     tekChainThrow(pushEvent(&event_queue, event));
301
302     return SUCCESS;
303 }
304
305 /**
306  * Delete a body snapshot from a scenario, and push a body
delete event to the event queue.
307  * @param scenario The scenario from which to delete the body.
308  * @param snapshot_id The ID of the snapshot to be deleted.
309  * @throws MEMORY_EXCEPTION if malloc() fails.
310  * @throws FAILURE if a snapshot with that ID was not found.
311 */
312 static exception tekDeleteBodySnapshot(TekScenario* scenario,
const int snapshot_id) {
313     // get snapshot by id
314     TekBodySnapshot* snapshot;
315     tekChainThrow(tekScenarioGetSnapshot(scenario,
(uint)snapshot_id, &snapshot));
316
317     // free model and material strings that got allocated
318     free(snapshot->model);
319     free(snapshot->material);
320
321     // delete snapshot from scenario
322     tekChainThrow(tekScenarioDeleteSnapshot(scenario,
(uint)snapshot_id));
323
324     // create event

```

```

325     TekEvent event = {};
326     event.type = BODY_DELETE_EVENT;
327     event.data.body.id = (uint)snapshot_id;
328
329     // push event to event queue
330     tekChainThrow(pushEvent(&event_queue, event));
331
332     return SUCCESS;
333 }
334
335 /**
336  * Recreate a body from a body snapshot. This should be called
337 if a large change happens to a body, such as changing the model so
338 that a new collision structure can be made.
339  * @param snapshot The body snapshot that should be used to
340 recreate the body.
341  * @param snapshot_id The ID of the snapshot that should be
342 recreated.
343  * @note Functionally this is the same as deleting the body
344 and making a new one with the same ID.
345  * @throws MEMORY_EXCEPTION if malloc() fails.
346 */
347 static exception tekRecreateBodySnapshot(const
TekBodySnapshot* snapshot, const int snapshot_id) {
348     // create event
349     TekEvent event = {};
350     event.type = BODY_DELETE_EVENT;
351     event.data.body.id = (uint)snapshot_id;
352     tekChainThrow(pushEvent(&event_queue, event));
353
354 /**
355  * Change the model of a snapshot body. This needs a separate
356 function as changing a model requires the physics body to be
357 recreated.
358  * @param scenario The scenario that contains the body
359 snapshot that needs a new model.
360  * @param snapshot_id The ID of the snapshot that needs to be
361 changed.

```

```

358     * @param model The filename of the new model.
359     * @throws MEMORY_EXCEPTION if realloc() fails.
360     * @throws FAILURE if a snapshot with that ID could not be
361     * found.
362
363     static exception tekChangeBodySnapshotModel(const TekScenario*
364 scenario, const int snapshot_id, const char* model) {
365         // get body snapshot
366         TekBodySnapshot* snapshot;
367         tekChainThrow(tekScenarioGetSnapshot(scenario,
368 (uint)snapshot_id, &snapshot));
369
370         // need to copy the model string, find length, allocate
371         // buffer and copy
372         const uint len_model = strlen(model) + 1;
373         char* new_model = realloc(snapshot->model, len_model);
374         if (!new_model)
375             tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
376 for new model.");
377         memcpy(new_model, model, len_model);
378         snapshot->model = new_model;
379
380         // push event to event queue
381         tekChainThrow(tekRecreateBodySnapshot(snapshot,
382 snapshot_id));
383
384         return SUCCESS;
385     }
386
387     /**
388      * Change the material of a snapshot body. This requires a
389      * separate function because the buffer containing the material
390      * filename needs to be reallocated.
391      * @param scenario The scenario that contains the body
392      * snapshot to be updated.
393      * @param snapshot_id The ID of the snapshot to be updated.
394      * @param material The filename of the material to be used.
395      * @throws MEMORY_EXCEPTION if realloc() fails.
396      * @throws FAILURE if a body with that ID could not be found.
397      */
398
399     static exception tekChangeBodySnapshotMaterial(const
400 TekScenario* scenario, const int snapshot_id, const char* material)
401 {

```

```

390     // get body snapshot by id
391     TekBodySnapshot* snapshot;
392     tekChainThrow(tekScenarioGetSnapshot(scenario,
393     (uint)snapshot_id, &snapshot));
394     // copy string in case original string gets deallocated.
395     const uint len_material = strlen(material) + 1;
396     char* new_material = realloc(snapshot->material,
397     len_material);
398     if (!new_material)
399         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for new material.");
400     memcpy(new_material, material, len_material);
401     snapshot->material = new_material;
402     // push event to event queue
403     tekChainThrow(tekRecreateBodySnapshot(snapshot,
404     snapshot_id));
405     return SUCCESS;
406 }
407
408 /**
409  * Reset the scenario so that there are no bodies left, and it
410  * is back to its original state.
411  * @param scenario The scenario to reset.
412  * @throws MEMORY_EXCEPTION if malloc() fails.
413 */
414 static exception tekResetScenario(const TekScenario* scenario)
{
415     // firstly, delete all bodies in the scenario
416     // create event
417     TekEvent event = {};
418     event.type = CLEAR_EVENT;
419     // push event to event queue
420     tekChainThrow(pushEvent(&event_queue, event));
421
422     // get all snapshot ids
423     uint* ids;
424     uint num_ids;

```

```

425     tekChainThrow(tekScenarioGetAllIds(scenario, &ids,
426     &num_ids));
427     // for each id, push a create event to rebuild the new
428     scenario
429     for (uint i = 0; i < num_ids; i++) {
430         const uint id = ids[i];
431         TekBodySnapshot* snapshot;
432         tekChainThrow(tekScenarioGetSnapshot(scenario, id,
433         &snapshot));
434         tekChainThrow(tekBodyCreateEvent(snapshot, id));
435     }
436     // cuz they got allocated before
437     free(ids);
438     return SUCCESS;
439 }
440
441 /**
442 * Restart a scenario, updating all bodies to the values
443 * specified by the scenario.
444 * @param scenario The scenario to use to reset the bodies
445 * back to their original positions.
446 * @throws MEMORY_EXCEPTION if malloc() fails.
447 */
448 static exception tekRestartScenario(const TekScenario*
449 scenario) {
450     // get all ids
451     uint* ids;
452     uint num_ids;
453     tekChainThrow(tekScenarioGetAllIds(scenario, &ids,
454     &num_ids));
455     // for each id, update scenario
456     for (uint i = 0; i < num_ids; i++) {
457         const uint id = ids[i];
458         tekChainThrow(tekUpdateBodySnapshot(scenario, id));
459     }
460     // ids got allocated so free them
461     free(ids);
462     return SUCCESS;

```

```

461  }
462
463  /**
464   * Push a time event to the event queue, which will change the
465   * rate and speed of the simulation.
466   * @param rate The rate of the simulation, the number of
467   * updates per second.
468   * @param speed The speed of the simulation relative to real
469   * life.
470   * @throws MEMORY_EXCEPTION if malloc() fails.
471   */
472 static exception tekSimulationSpeedEvent(const double rate,
473 const double speed, TekGuiOptionWindow* runner_window) {
474     // create an event
475     TekEvent event = {};
476     event.type = TIME_EVENT;
477     event.data.time.rate = rate;
478     event.data.time.speed = speed;
479
480     // push event to event queue
481     tekChainThrow(pushEvent(&event_queue, event));
482
483     // update gui to show new speed and rate
484     tekChainThrow(tekGuiWriteNumberOption(runner_window,
485 "rate", rate));
486     tekChainThrow(tekGuiWriteNumberOption(runner_window,
487 "speed", speed));
488
489     return SUCCESS;
490 }
491 /**
492  * Push a gravity event to the event queue, which will change
493  * the acceleration due to gravity.
494  * @param gravity The new acceleration due to gravity.
495  * @param runner_window The runner window to update with the
496  * new value.
497  * @throws MEMORY_EXCEPTION if malloc() fails.
498  */
499 static exception tekGravityEvent(const double gravity,
500 TekGuiOptionWindow* runner_window) {

```

```

494     // create event
495     TekEvent event = {};
496     event.type = GRAVITY_EVENT;
497     event.data.gravity = (float)gravity;
498
499     // push event to event queue
500     tekChainThrow(pushEvent(&event_queue, event));
501
502     // write new gravity to the gui
503     tekChainThrow(tekGuiWriteNumberOption(runner_window,
504 "gravity", gravity));
505
506 }
507
508 /**
509  * Push a pause event to the event queue, which will stop the
simulation temporarily.
510  * @param paused 1 if paused, 0 if not
511  * @throws MEMORY_EXCEPTION if malloc() fails.
512 */
513 static exception tekPauseEvent(const flag paused) {
514     // create event
515     TekEvent event = {};
516     event.type = PAUSE_EVENT;
517     event.data.paused = paused;
518
519     // push event to event queue
520     tekChainThrow(pushEvent(&event_queue, event));
521
522     return SUCCESS;
523 }
524
525 /**
526  * Hide all the possible windows from each menu mode.
527  * @param gui The gui components.
528 */
529 static void tekHideAllWindows(struct TekGuiComponents* gui) {
530     // do i really need to explain this?
531     // visible = 0 means not visible.
532     gui->hierarchy_window.window.visible = 0;
533     gui->editor_window.window.visible = 0;
534     gui->action_window.window.visible = 0;

```

```

535     gui->scenario_window.window.visible = 0;
536     gui->save_window.window.visible = 0;
537     gui->load_window.window.visible = 0;
538     gui->runner_window.window.visible = 0;
539     gui->inspect_window.window.visible = 0;
540 }
541 /**
542  * Switches the menu mode into the main menu mode. Brings the
543 start button to the front.
544  * @param gui The gui components.
545  * @throws LIST_EXCEPTION .
546 */
547 static exception tekSwitchToMainMenu(struct TekGuiComponents*
gui) {
548     // bring button to front so it isn't obscured by other
buttons
549     tekChainThrow(tekGuiBringButtonToFront(&gui-
>start_button.button));
550
551     // hide everything else
552     tekHideAllWindows(gui);
553     return SUCCESS;
554 }
555
556 /**
557  * Switches the menu mode into the builder menu mode. Makes
all relevant windows visible and brings them to the front.
558  * @param gui The gui components.
559  * @throws MEMORY_EXCEPTION if malloc() fails.
560 */
561 static exception tekSwitchToBuilderMenu(struct
TekGuiComponents* gui) {
562     // hide all windows
563     tekHideAllWindows(gui);
564
565     // show the relevant windows for this menu
566     gui->hierarchy_window.window.visible = 1;
567     gui->action_window.window.visible = 1;
568     gui->scenario_window.window.visible = 1;
569
570     // bring relevant windows to front

```

```

571     tekChainThrow(tekGuiBringWindowToFront(&gui-
>hierarchy_window.window));
572     tekChainThrow(tekGuiBringWindowToFront(&gui-
>action_window.window));
573     tekChainThrow(tekGuiBringWindowToFront(&gui-
>editor_window.window));
574     tekChainThrow(tekGuiBringWindowToFront(&gui-
>scenario_window.window));
575
576     // restart scenario, so that all the bodies are in their
starting positions
577     tekChainThrow(tekRestartScenario(&active_scenario));
578
579     // read the begin paused value
580     flag begin_paused;
581     tekChainThrow(tekGuiReadBooleanOption(&gui-
>scenario_window, "pause", &begin_paused));
582
583     // if begin paused, then the simulation should begin
paused, so pre-empt the pause event.
584     tekChainThrow(tekPauseEvent(begin_paused));
585
586     return SUCCESS;
587 }
588
589 /**
590 * Switches the menu mode to the save menu mode. Brings the
save window to the front and re-centres it.
591 * @param gui The gui components.
592 * @throws MEMORY_EXCEPTION if malloc() fails.
593 */
594 static exception tekSwitchToSaveMenu(struct TekGuiComponents*
gui) {
595     // hide all windows
596     tekHideAllWindows(gui);
597
598     // show only the save window
599     gui->save_window.window.visible = 1;
600
601     // get the size of the save window
602     int window_width, window_height;
603     tekGetWindowSize(&window_width, &window_height);
604

```

```

605     // change the window position so its centred.
606     tekGuiSetWindowPosition(
607         &gui->save_window.window,
608         (window_width - (int)gui->save_window.window.width) /
2,
609         (window_height - (int)gui-
610             >save_window.window.height) / 2
611     );
612     // bring it to the front so other buttons dont go on top
of it.
613     tekChainThrow(tekGuiBringWindowToFront(&gui-
>save_window.window));
614     return SUCCESS;
615 }
616
617 /**
618 * Switches the menu mode to the load menu mode. Brings the
load window to the front and re-centres it.
619 * @param gui The gui components.
620 * @throws MEMORY_EXCEPTION if malloc() fails.
621 */
622 static exception tekSwitchToLoadMenu(struct TekGuiComponents*
gui) {
623     // hide all windows
624     tekHideAllWindows(gui);
625
626     // show only the load window
627     gui->load_window.window.visible = 1;
628
629     // get size of load window
630     int window_width, window_height;
631     tekGetWindowSize(&window_width, &window_height);
632
633     // update position so it's centred
634     tekGuiSetWindowPosition(
635         &gui->load_window.window,
636         (window_width - (int)gui->load_window.window.width) /
2,
637         (window_height - (int)gui-
638             >load_window.window.height) / 2
639     );

```

```

640      // bring window to the front
641      tekChainThrow(tekGuiBringWindowToFront(&gui-
>load_window.window));
642      return SUCCESS;
643  }
644
645 /**
646  * Switches the menu mode to the runner menu mode. Displays
the runner window and brings it to the front.
647  * @param gui The gui components.
648  * @throws LIST_EXCEPTION if there was a cosmic bit flip that
changed a variable somewhere.
649 */
650 static exception tekSwitchToRunnerMenu(struct
TekGuiComponents* gui) {
651     // hide all windows
652     tekHideAllWindows(gui);
653
654     // set runner window and inspect window to be visible
655     gui->runner_window.window.visible = 1;
656     gui->inspect_window.visible = 1;
657
658     // bring both windows to the front
659     tekChainThrow(tekGuiBringWindowToFront(&gui-
>runner_window.window));
660     tekChainThrow(tekGuiBringWindowToFront(&gui-
>inspect_window));
661
662     // reset inspect index in case the inspect index points to
an object that has since been deleted
663     inspect_index = -1;
664     tekChainThrow(tekPushInspectEvent(0));
665
666     // restart again because sometimes the velocities remain.
667     tekChainThrow(tekRestartScenario(&active_scenario));
668
669     return SUCCESS;
670 }
671
672 /**
673  * Switches the menu mode based on what the next menu mode is
specified to be. Resets the next mode to -1.
674  * @param gui The gui components.

```

```

675  * @note Uses the static variables mode and next_mode, this
allows different callbacks to edit the mode.
676  * @throws MEMORY_EXCEPTION if malloc() fails.
677  */
678 static exception tekChangeMenuMode(struct TekGuiComponents*
gui) {
679     // create event
680     TekEvent event = {};
681     event.type = MODE_CHANGE_EVENT;
682     event.data.mode = next_mode;
683
684     // push event to event queue
685     tekChainThrow(pushEvent(&event_queue, event));
686
687     /// basically check which mode it is and run the correct
change function
688     switch (next_mode) {
689         case MODE_MAIN_MENU:
690             tekChainThrow(tekSwitchToMainMenu(gui));
691             break;
692         case MODE_BUILDER:
693             tekChainThrow(tekSwitchToBuilderMenu(gui));
694             break;
695         case MODE_RUNNER:
696             tekChainThrow(tekSwitchToRunnerMenu(gui));
697             break;
698         case MODE_SAVE:
699             tekChainThrow(tekSwitchToSaveMenu(gui));
700             break;
701         case MODE_LOAD:
702             tekChainThrow(tekSwitchToLoadMenu(gui));
703             break;
704         default:
705             // if the mode is not recognised, then dont do
anything.
706             next_mode = -1;
707             return SUCCESS;
708     }
709
710     // update the mode
711     mode = next_mode;
712     next_mode = -1;
713     return SUCCESS;

```

```

714 }
715 /**
716 * The callback for the start button. If the button was left-
717 clicked, the mode is changed to builder.
718 * @param button The start button / button being pressed.
719 * @param callback_data Callback data from this button such as
720 mouse position.
721 */
722 static void tekStartButtonCallback(TekGuiTextButton* button,
TekGuiButtonCallbackData callback_data) {
723     // make sure we are clicking the left button in main menu
mode
724     if (mode != MODE_MAIN_MENU)
725         return;
726     if (callback_data.type !=
727 TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK)
728         return;
729     if (callback_data.data.mouse_button.action != GLFW_RELEASE
|| callback_data.data.mouse_button.button != GLFW_MOUSE_BUTTON_LEFT)
730         return;
731
732     // if so, change to builder mode.
733     next_mode = MODE_BUILDER;
734 }
735 /**
736 * Create the main menu, this includes creating the logo, the
start button and the splash text.
737 * @param window_width The width of the window.
738 * @param window_height The height of the window.
739 * @param gui The gui components.
740 * @throws MEMORY_EXCEPTION if malloc() fails.
741 * @throws FILE_EXCEPTION if the logo couldn't be found.
742 */
743 static exception tekCreateMainMenu(const int window_width,
const int window_height, struct TekGuiComponents* gui) {
744     // create start button
745     tekChainThrow(tekGuiCreateTextButton("", &gui-
>start_button));
746
747

```

```

748     // want larger text than default, so update the text
height and recreate the text
749     gui->start_button.text_height = 25;
750     tekChainThrow(tekGuiSetTextButtonText(&gui->start_button,
"Start"));
751     tekChainThrow(tekGuiSetTextButtonSize(&gui->start_button,
START_BUTTON_WIDTH, START_BUTTON_HEIGHT));
752     tekChainThrow(tekGuiSetTextButtonPosition(&gui-
>start_button, (window_width - START_BUTTON_WIDTH) / 2 ,
(window_height - START_BUTTON_HEIGHT) / 2));
753     gui->start_button.text_height = 50;
754     gui->start_button.callback = tekStartButtonCallback;
755
756     // create the tekphysics logo texture/image
757     tekChainThrow(tekGuiCreateImage(
758         LOGO_WIDTH, LOGO_HEIGHT,
759         "../res/tekphysics.png",
760         &gui->logo
761     ));
762     return SUCCESS;
763 }
764
765 /**
766 * Draw the main menu. Draws the logo, start button and splash
text.
767 * @param gui The gui components.
768 * @throws MEMORY_EXCEPTION if malloc() fails.
769 * @throws OPENGL_EXCEPTION if OpenGL fails.
770 * @throws VECTOR_EXCEPTION if a vector is written out of
range.
771 */
772 static exception tekDrawMainMenu(struct TekGuiComponents* gui)
{
773     // get window size for a variety of things later on ...
774     int window_width, window_height;
775     tekGetWindowSize(&window_width, &window_height);
776
777     // draw start button
778     tekChainThrow(tekGuiDrawTextButton(&gui->start_button));
779
780     // get logo position and draw it
781     const float x = ((float)window_width - LOGO_WIDTH) / 2.0f;

```

```

782     const float y = (float)window_height / 4.0f -
LOGO_HEIGHT / 2.0f;
783
784     tekChainThrow(tekGuiDrawImage(&gui->logo, x, y));
785     tekChainThrow(tekGuiSetTextButtonPosition(&gui-
>start_button, (window_width - START_BUTTON_WIDTH) / 2 ,
(window_height - START_BUTTON_HEIGHT) / 2));
786
787     // get current time so we can make the splash text wiggle
at a certain rate
788     struct timespec curr_time;
789     clock_gettime(CLOCK_MONOTONIC, &curr_time);
790
791     // make a full swing every 4 seconds
792     const double progression = (double)(curr_time.tv_sec % 4)
+ (double)curr_time.tv_nsec / (double)BILLION;
793     const double angle = progression * CGLM_PI_2;
794     const double sin_angle = sin(angle);
795     const double max_angle = CGLM_PI / 12.0; // max angle =
pi/12
796
797     // draw the splash text
798     tekChainThrow(tekDrawColouredRotatedText(
799         &gui->splash_text,
800         x + LOGO_WIDTH - gui->splash_text.width / 2.0f, y +
LOGO_HEIGHT - gui->splash_text.height / 2.0,
801         (vec4){1.0f, 1.0f, 0.0f, 1.0f},
802         x + LOGO_WIDTH,
803         y + LOGO_HEIGHT,
804         sin_angle * max_angle
805     ));
806
807     return SUCCESS;
808 }
809
810 /**
811  * Delete the main menu, clearing any memory from the gui
components used.
812  * @param gui The gui components.
813  */
814 static void tekDeleteMainMenu(const struct TekGuiComponents*
gui) {
815     // delete everything we allocated before

```

```

816     tekGuiDeleteTextButton(&gui->start_button);
817     tekGuiDeleteImage(&gui->logo);
818     tekDeleteText(&gui->splash_text);
819 }
820
821 /**
822  * Update the editor window with the current values of a body
823  * snapshot.
824  * @param editor_window The editor window which to update.
825  * @param scenario The scenario that contains the body
826  * snapshot.
827  * @note Uses the currently selected object in the hierarchy
828  * window as the ID.
829  * @throws LIST_EXCEPTION if the currently selected ID doesn't
830  * exist
831  * @throws HASHTABLE_EXCEPTION if one of the option keys
832  * doesn't exist.
833 */
834 static exception tekUpdateEditorWindow(TekGuiOptionWindow*
835 editor_window, const TekScenario* scenario) {
836     if (hierarchy_index < 0) {
837         editor_window->window.visible = 0;
838         return SUCCESS;
839     }
840
841     // read information from the selected object
842     editor_window->window.visible = 1;
843     TekBodySnapshot* body;
844     tekChainThrow(tekScenarioGetSnapshot(scenario,
845 (uint)hierarchy_index, &body));
846     char* object_name;
847
848     // read the name separately because not in the snapshot
849     data
850     tekChainThrow(tekScenarioGetName(scenario,
851 (uint)hierarchy_index, &object_name));
852     tekChainThrow(tekGuiWriteStringOption(editor_window,
853 "name", object_name, strlen(object_name) + 1));
854
855     // write all the values into the editor
856     tekChainThrow(tekGuiWriteVec3Option(editor_window,
857 "position", body->position));

```

```

847      tekChainThrow(tekGuiWriteVec3Option(editor_window,
"rotation", body->rotation));
848      tekChainThrow(tekGuiWriteVec3Option(editor_window,
"velocity", body->velocity));
849      tekChainThrow(tekGuiWriteNumberOption(editor_window,
"mass", body->mass));
850      tekChainThrow(tekGuiWriteNumberOption(editor_window,
"friction", body->friction));
851      tekChainThrow(tekGuiWriteNumberOption(editor_window,
"restitution", body->restitution));
852      tekChainThrow(tekGuiWriteBooleanOption(editor_window,
"immovable", (flag)body->immovable));
853      tekChainThrow(tekGuiWriteStringOption(editor_window,
"model", body->model, strlen(body->model) + 1));
854      tekChainThrow(tekGuiWriteStringOption(editor_window,
"material", body->material, strlen(body->material) + 1));
855
856      return SUCCESS;
857 }
858
859 /**
860  * The callback for any updates to the hierarchy window.
Updates the currently selected body snapshot.
861  * @param hierarchy_window The list window that was updated /
the hierarchy window.
862 */
863 static void tekHierarchyCallback(TekGuiListWindow*
hierarchy_window) {
864     if (mode != MODE_BUILDER)
865         return;
866     TekGuiOptionWindow* editor_window = hierarchy_window-
>data;
867     // all callbacks mean that an object was selected in the
hierarchy
868
869     // get the index of the clicked object
870     int index;
871     tekScenarioGetByNameIndex(&active_scenario,
hierarchy_window->select_index, NULL, &index);
872
873     // if index == -1, then the user clicked to create a new
object
874     if (index == -1) {

```

```

875         tekCreateBodySnapshot(&active_scenario,
&hierarchy_index);
876     } else {
877         // otherwise, an existing one was selected
878         hierarchy_index = index;
879     }
880
881     // update editor window to show the object's properties in
there
882     tekUpdateEditorWindow(editor_window, &active_scenario);
883 }
884
885 /**
886 * Set a string input to display an error.
887 * @param window The option window to apply the function to.
888 * @param key The name of the option.
889 * @param error_message The error message to write into the
option.
890 * @throws MEMORY_EXCEPTION if malloc() fails.
891 */
892 static exception tekDisplayOptionError(TekGuiOptionWindow*
window, const char* key, const char* error_message) {
893     // wrapper around write option, calculate length of
string.
894     tekChainThrow(tekGuiWriteStringOption(window, key,
error_message, strlen(error_message) + 1))
895     return SUCCESS;
896 }
897
898 /**
899 * Read a filename for a new, unwritten file. Checks to make
sure the input is not empty, but will allow any filename otherwise.
900 * If the inputted filename ends with the requested extension,
it is not appended.
901 * @param window The window to read the input from.
902 * @param key The key of where to find the input data.
903 * @param directory The base directory of where the file
should be stored.
904 * @param extension The extension that should be added to the
file if not already added.
905 * @param filepath The outputted filepath
906 * @note "filepath" is allocated, so needs to be freed by the
caller.

```

```

907     * @throws MEMORY_EXCEPTION if malloc() fails.
908     */
909     static exception tekReadNewFileInput(TekGuiOptionWindow*
window, const char* key, const char* directory, const char*
extension, char** filepath) {
910         // get inputted file from the user
911         char* filename;
912         tekChainThrow(tekGuiReadStringOption(window, key,
&filename));
913
914         // if nothing inputted, remind user to write something
915         if (!filename || !filename[0]) {
916             *filepath = 0;
917             return tekDisplayOptionError(window, key, "ENTER
FILE");
918         }
919
920         // get absolute path
921         tekChainThrow(addPathToFile(directory, filename,
filepath));
922
923         // checking if the file already has an extension on it
924         const uint len_filename = strlen(filename);
925         const uint len_extension = strlen(extension);
926         flag needs_ext = 1;
927         if (len_filename >= len_extension) {
928             const char* ext_ptr = filename + len_filename - 5;
929             if (!strcmp(ext_ptr, extension)) {
930                 needs_ext = 0;
931             }
932         }
933
934         // append the extension if not there
935         if (needs_ext) {
936             char* filepath_ext;
937             tekChainThrow(addPathToFile(*filepath, ".tscn",
&filepath_ext));
938             free(*filepath);
939             *filepath = filepath_ext;
940         }
941
942         return SUCCESS;
943     }

```

```

944
945  /**
946   * Get the name of an existing file. The function will reject
any inputs if the file specified does not exist, or if the input
string is empty.
947   * If a filename exists with the same name but the real file
includes the specified extension, then this input is accepted.
948   * @param window The window which contains the file input.
949   * @param key The name of the option which has the input.
950   * @param directory The base directory of the file.
951   * @param extension The extension to be added to the file.
952   * @param filepath The outputted filepath.
953   * @note The filepath is allocated, and needs to be freed by
the caller.
954   * @throws MEMORY_EXCEPTION if malloc() fails.
955  */
956 static exception tekReadExistingFileInput(TekGuiOptionWindow*
window, const char* key, const char* directory, const char*
extension, char** filepath) {
957     // read the inputted filename
958     char* filename;
959     tekChainThrow(tekGuiReadStringOption(window, key,
&filename));
960
961     // if filename empty, remind user to write something
962     if (!filename || !filename[0]) {
963         *filepath = 0;
964         return tekDisplayOptionError(window, key, "ENTER
FILE");
965     }
966
967     // prevent direct access to filesystem
968     if (fileExists(filename)) {
969         *filepath = 0;
970         return SUCCESS;
971     }
972
973     // build absolute path
974     tekChainThrow(addPathToFile(directory, filename,
filepath));
975
976     // if file isnt valid right now
977     if (!fileExists(*filepath)) {

```

```

978         // check if they entered the file without the
extension
979         char* filepath_ext;
980         tekChainThrow(addPathToFile(*filepath, extension,
&filepath_ext));
981         free(*filepath);
982         if (!fileExists(filepath_ext)) {
983             free(filepath_ext);
984             *filepath = 0;
985             return tekDisplayOptionError(window, key, "NOT
FOUND");
986         }
987         *filepath = filepath_ext;
988     }
989
990     return SUCCESS;
991 }
992
993 /**
994 * The callback for the editor window. This is called whenever
a user updates a body snapshot's properties or deletes it.
995 * @param window The option window / editor window in this
case.
996 * @param callback_data The callback data containing the name
of the option and its type + data.
997 * @throws MEMORY_EXCEPTION if malloc() fails.
998 */
999 static exception tekEditorCallback(TekGuiOptionWindow* window,
TekGuiOptionWindowCallbackData callback_data) {
1000     // shouldnt be able to edit while not in builder mode
1001     if (mode != MODE_BUILDER)
1002         return SUCCESS;
1003
1004     // if index < 0, then no object is selected to edit.
1005     if (hierarchy_index < 0)
1006         return SUCCESS;
1007
1008     // if delete selected, then delete this object
1009     if (!strcmp(callback_data.name, "delete")) {
1010         tekChainThrow(tekDeleteBodySnapshot(&active_scenario,
hierarchy_index));
1011         hierarchy_index = -1; // set index to -1 because this
object cannot be selected if it doesnt exist.

```

```

1012         return SUCCESS;
1013     }
1014
1015     // updating the name
1016     if (!strcmp(callback_data.name, "name")) {
1017         // read what the user inputted and tell scenario to
1018         // update this name
1019         char* inputted_name;
1020         tekChainThrow(tekGuiReadStringOption(window, "name",
1021             &inputted_name));
1022         tekChainThrow(tekScenarioSetName(&active_scenario,
1023             (uint)hierarchy_index, inputted_name));
1024         return SUCCESS;
1025     }
1026
1027     // updating the model file
1028     if (!strcmp(callback_data.name, "model")) {
1029         // check if we have a valid file
1030         char* filepath;
1031         tekChainThrow(tekReadExistingFileInput(window,
1032             "model", "../res/", ".tmsh", &filepath));
1033         if (!filepath)
1034             return SUCCESS;
1035         free(filepath);
1036
1037         return SUCCESS;
1038     }
1039
1040     // updating the material file
1041     if (!strcmp(callback_data.name, "material")) {
1042         // check if we have a valid file
1043         char* filepath;
1044         tekChainThrow(tekReadExistingFileInput(window,
1045             "material", "../res/", ".tmat", &filepath));
1046         if (!filepath)
1047             return SUCCESS;

```

```

1048         // update the material
1049
tekChainThrow(tekChangeBodySnapshotMaterial(&active_scenario,
hierarchy_index, filepath));
1050
1051         free(filepath);
1052
1053         return SUCCESS;
1054     }
1055
1056         // after this, other options are eliminated, we must be
changing one of the properties
1057
1058         // get snapshot from scenario, we are going to update
everything now
1059     TekBodySnapshot* snapshot;
1060     tekChainThrow(tekScenarioGetSnapshot(&active_scenario,
(uint)hierarchy_index, &snapshot));
1061
1062         // copy in locational things
1063     tekChainThrow(tekGuiReadVec3Option(window, "position",
snapshot->position));
1064     tekChainThrow(tekGuiReadVec3Option(window, "rotation",
snapshot->rotation));
1065     tekChainThrow(tekGuiReadVec3Option(window, "velocity",
snapshot->velocity));
1066
1067         // copy in properties
1068     double number;
1069     tekChainThrow(tekGuiReadNumberOption(window, "mass",
&number));
1070     snapshot->mass = (float)number;
1071     if (snapshot->mass < 0.0001f)
1072         snapshot->mass = 0.0001f;
1073     tekChainThrow(tekGuiReadNumberOption(window,
"restitution", &number));
1074     snapshot->restitution = (float)number;
1075     if (snapshot->restitution > 1.0f)
1076         snapshot->restitution = 1.0f;
1077     if (snapshot->restitution < 0.0f)
1078         snapshot->restitution = 0.0f;
1079     tekChainThrow(tekGuiReadNumberOption(window, "friction",
&number));

```

```

1080     snapshot->friction = (float)number;
1081     if (snapshot->friction > 1.0f)
1082         snapshot->friction = 1.0f;
1083     if (snapshot->friction < 0.0f)
1084         snapshot->friction = 0.0f;
1085
1086     flag immovable;
1087     tekChainThrow(tekGuiReadBooleanOption(window, "immovable",
&immovable));
1088     snapshot->immovable = (int)immovable;
1089
1090     // push changes
1091     tekChainThrow(tekUpdateEditorWindow(window,
&active_scenario));
1092     tekChainThrow(tekUpdateBodySnapshot(&active_scenario,
hierarchy_index));
1093
1094     return SUCCESS;
1095 }
1096
1097 /**
1098 * The callback for the action window. Called whenever the
user selects an option like "save", "load" "run" or "quit".
1099 * @param window The list window / action window.
1100 */
1101 static void tekActionCallback(TekGuiListWindow* window) {
1102     // action menu not visible in other modes.
1103     if (mode != MODE_BUILDER)
1104         return;
1105
1106     // based on the index of what we clicked, change the menu
mode
1107     switch (window->select_index) {
1108     case SAVE_OPTION:
1109         next_mode = MODE_SAVE;
1110         break;
1111     case LOAD_OPTION:
1112         next_mode = MODE_LOAD;
1113         break;
1114     case RUN_OPTION:
1115         next_mode = MODE_RUNNER;
1116         break;

```

```

1117     case QUIT_OPTION: // quit doesnt exit the program, the X
button does that :D
1118         next_mode = MODE_MAIN_MENU;
1119         break;
1120     default:
1121         return;
1122     }
1123 }
1124 /**
1125 * Create the list of possible actions for the action window.
1126 * @param actions_list A pointer to a list of actions that
will be allocated. This needs to be freed.
1127 * @throws MEMORY_EXCEPTION if malloc() fails.
1128 */
1129 static exception tekCreateActionsList(List** actions_list) {
1130     // allocate memory for the list and create it
1131     *actions_list = (List*)malloc(sizeof(List));
1132     if (!*actions_list)
1133         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for action list.");
1134     listCreate(*actions_list);
1135
1136     // for each option, create the option
1137     for (uint i = 0; i < NUM_OPTIONS; i++) {
1138         const char* string;
1139         switch (i) {
1140             // do it like this, so that the index in the list
correlates with the value of the macro.
1141             // then i can change the macros and it works still.
1142             // and i can do if index == SAVE_OPTION and it looks
readable
1143             case SAVE_OPTION:
1144                 string = "Save";
1145                 break;
1146             case LOAD_OPTION:
1147                 string = "Load";
1148                 break;
1149             case RUN_OPTION:
1150                 string = "Run";
1151                 break;
1152             case QUIT_OPTION:
1153                 string = "Quit";

```

```

1155             break;
1156         default:
1157             tekThrow(FAILURE, "Unknown option id in action
menu.");
1158         }
1159
1160         // create copy of string and put into list
1161         const uint len_string = strlen(string) + 1;
1162         char* buffer = (char*)malloc(sizeof(char) *
len_string);
1163         memcpy(buffer, string, len_string);
1164         tekChainThrow(listAddItem(*actions_list, buffer));
1165     }
1166
1167     return SUCCESS;
1168 }
1169
1170 /**
1171 * The callback for the scenario options window. Called when
options such as gravity, sky colour and start paused are changed.
1172 * @param window The option window / scenario window.
1173 * @param callback_data The callback data including the option
name that was changed and the type of data.
1174 * @throws MEMORY_EXCEPTION if malloc() fails.
1175 */
1176 static exception tekScenarioCallback(TekGuiOptionWindow*
window, TekGuiOptionWindowCallbackData callback_data) {
1177     // if callback has edited gravity value
1178     if (!strcmp(callback_data.name, "gravity")) {
1179         // clamp value to reasonable ones
1180         double gravity;
1181         tekChainThrow(tekGuiReadNumberOption(window,
"gravity", &gravity));
1182         if (gravity < 0.0) {
1183             gravity = 0.0;
1184         }
1185         if (gravity > 100.0) {
1186             gravity = 100.0;
1187         }
1188         // push event
1189         tekChainThrow(tekGravityEvent(gravity, window));
1190         return SUCCESS;
1191     }

```

```

1192
1193     if (!strcmp(callback_data.name, "pause")) { // if pause
event
1194         // this is the option of whether the game should begin
in a paused state
1195         flag begin_paused;
1196         tekChainThrow(tekGuiReadBooleanOption(window, "pause",
&begin_paused));
1197         tekChainThrow(tekPauseEvent(begin_paused));
1198     }
1199
1200     if (!strcmp(callback_data.name, "sky_colour")) { // sky-
colour event
1201         // update sky colour value with inputted value
1202         vec3 sky_colour;
1203         tekChainThrow(tekGuiReadVec3Option(window,
"sky_colour", sky_colour));
1204         tekSetWindowColour(sky_colour);
1205     }
1206
1207     return SUCCESS;
1208 }
1209
1210 /**
1211 * Create the builder menu, this includes creating all the
windows related to the object hierarchy, the options menu, the
editor window and the scenario window.
1212 * @param window_width The width of the window.
1213 * @param window_height The height of the window.
1214 * @param gui The gui components.
1215 * @throws MEMORY_EXCEPTION if malloc() fails.
1216 */
1217 static exception tekCreateBuilderMenu(const int window_width,
const int window_height, struct TekGuiComponents* gui) {
1218     // create an empty scenario
1219     tekCreateScenario(&active_scenario);
1220
1221     // create a list window for the object hierarchy
1222     // link to the scenario name list.
1223     tekChainThrow(tekGuiCreateListWindow(&gui-
>hierarchy_window, &active_scenario.names));
1224     gui->hierarchy_window.data = &gui->editor_window;
1225     gui->hierarchy_window.callback = tekHierarchyCallback;

```

```

1226     tekChainThrow(tekGuiSetWindowTitle(&gui-
>hierarchy_window.window, "Hierarchy"));
1227     tekGuiSetWindowPosition(&gui->hierarchy_window.window, 10,
10 + (int)gui->hierarchy_window.window.title_width);
1228
1229     // create editor option window, allow for position, mass
and other properties to be edited
1230
tekChainThrow(tekGuiCreateOptionWindow("../res/windows/editor.yml",
&gui->editor_window))
1231     gui->editor_window.window.visible = 0;
1232     gui->editor_window.data = NULL;
1233     gui->editor_window.callback = tekEditorCallback;
1234
1235     // actions list, save, load, run and stuff
1236     List* actions_list;
1237     tekChainThrow(tekCreateActionsList(&actions_list));
1238     gui->action_window.callback = tekActionCallback;
1239     tekChainThrow(tekGuiCreateListWindow(&gui->action_window,
actions_list));
1240     tekChainThrow(tekGuiSetWindowTitle(&gui-
>action_window.window, "Actions"))
1241     tekGuiSetWindowPosition(&gui->action_window.window, 10,
(int)(gui->hierarchy_window.window.height + gui-
>hierarchy_window.window.title_width + gui-
>action_window.window.title_width) + 20);
1242
1243     // simulation options window
1244
tekChainThrow(tekGuiCreateOptionWindow("../res/windows/scenario.yml",
, &gui->scenario_window));
1245     tekChainThrow(tekGuiWriteNumberOption(&gui-
>scenario_window, "gravity", 9.81));
1246     tekChainThrow(tekGuiWriteBooleanOption(&gui-
>scenario_window, "pause", 0));
1247
1248     // sky colour, dont change tho cuz its ugly
1249     vec3 sky_colour = {0.3f, 0.3f, 0.3f};
1250     tekChainThrow(tekGuiWriteVec3Option(&gui->scenario_window,
"sky_colour", sky_colour));
1251     tekSetWindowColour(sky_colour);
1252
1253     gui->scenario_window.callback = tekScenarioCallback;

```

```

1254     gui->scenario_window.data = &gui->runner_window;
1255
1256     return SUCCESS;
1257 }
1258
1259 /**
1260  * Draw the builder menu. Draws all visible windows, and hides
1261  * the editor window if the hierarchy index is less than 0.
1262  * @param gui The gui components.
1263  * @throws OPENGL_EXCEPTION if there was a rendering error.
1264  */
1265 static exception tekDrawBuilderMenu(struct TekGuiComponents*
1266 gui) {
1267     if (hierarchy_index < 0) {
1268         gui->editor_window.window.visible = 0;
1269     }
1270
1271     // this does it all for us, the windows we dont want are
1272     // hidden.
1273     tekChainThrow(tekGuiDrawAllWindows());
1274     return SUCCESS;
1275 }
1276
1277 /**
1278  * Delete the builder menu, freeing any memory that was
1279  * allocated by its gui components.
1280  * @param gui The gui components.
1281  */
1282 static void tekDeleteBuilderMenu(struct TekGuiComponents* gui)
1283 {
1284     // delete everything we allocated for the builder menu
1285     tekGuiDeleteListWindow(&gui->hierarchy_window);
1286     tekGuiDeleteOptionWindow(&gui->editor_window);
1287     listFreeAllData(gui->action_window.text_list);
1288     listDelete(gui->action_window.text_list);
1289     tekGuiDeleteListWindow(&gui->action_window);
1290     tekGuiDeleteOptionWindow(&gui->scenario_window);
1291 }
1292
1293 /**
1294  * The callback for the save window. Called when the user
1295  * updates the save location or presses the save button.
1296  * @param window The option window / save window.

```

```

1291     * @param callback_data The callback data containing the name
1292     of the edited option and its type.
1293
1294     * @throws MEMORY_EXCEPTION if malloc() fails.
1295
1296     */
1297
1298     static exception tekSaveCallback(TekGuiOptionWindow* window,
1299     TekGuiOptionWindowCallbackData callback_data) {
1300         // non button callback = user is typing, so ignore that
1301         // event
1302         if (callback_data.type != TEK_BUTTON_INPUT) {
1303             return SUCCESS;
1304         }
1305
1306         // cancel = dont do anything, just quit
1307         if (!strcmp(callback_data.name, "cancel")) {
1308             next_mode = MODE_BUILDER;
1309             return SUCCESS;
1310         }
1311
1312         // if none of the above, the save button is being pressed
1313         // get file input from user
1314         char* filepath;
1315         tekChainThrow(tekReadNewFileInput(window, "filename",
1316         "./saves/", ".tscn", &filepath))
1317         if (filepath) { // if filepath is valid, save the scenario
1318             at that path
1319             tekChainThrow(tekWriteScenario(&active_scenario,
1320             filepath));
1321             next_mode = MODE_BUILDER;
1322             free(filepath);
1323         }
1324
1325         return SUCCESS;
1326     }
1327
1328
1329     /**
1330     * The callback for the load window. Called when the user
1331     updates the load location or presses the load button.
1332     * @param window The option window / load window.
1333     * @param callback_data The callback data containing the name
1334     of the edited option and its type.
1335     * @throws MEMORY_EXCEPTION if malloc() fails.
1336     */

```

```

1325 static exception tekLoadCallback(TekGuiOptionWindow* window,
TekGuiOptionWindowCallbackData callback_data) {
1326     // callback sent for every event in the window
1327     // if not a button input, we dont care, just the user
typing in their filename
1328     if (callback_data.type != TEK_BUTTON_INPUT) {
1329         return SUCCESS;
1330     }
1331
1332     // cancel = dont load any file
1333     if (!strcmp(callback_data.name, "cancel")) {
1334         next_mode = MODE_BUILDER;
1335         return SUCCESS;
1336     }
1337
1338     // otherwise, we have just pressed the load button, so get
the filepath of the file
1339     char* filepath;
1340     tekChainThrow(tekReadExistingFileInput(window, "filename",
"..../saves/", ".tscn", &filepath));
1341
1342     // if filepath == null, there was an invalid file.
1343     if (filepath) {
1344         // valid filepath = load that scenario file
1345         tekDeleteScenario(&active_scenario);
1346         tekChainThrow(tekReadScenario(filepath,
&active_scenario));
1347         tekChainThrow(tekResetScenario(&active_scenario));
1348
1349         free(filepath);
1350
1351         next_mode = MODE_BUILDER;
1352     }
1353
1354     return SUCCESS;
1355 }
1356
1357 /**
1358 * The callback for the runner window, called when user
updates the speed or rate of the simulation, or when they press
play, pause, stop or step.
1359 * @param window The option window / runner window.

```

```

1360     * @param callback_data The callback data containing the name
1361     of the option and its type.
1362
1363     * @throws MEMORY_EXCEPTION if malloc() fails.
1364
1365     */
1366
1367     static exception tekRunnerCallback(TekGuiOptionWindow* window,
1368     TekGuiOptionWindowCallbackData callback_data) {
1369         // callback for any event in the option window
1370         // check which button has been clicked
1371         // these three are simple, just do something and quit
1372         if (!strcmp(callback_data.name, "stop")) {
1373             next_mode = MODE_BUILDER;
1374             return SUCCESS;
1375         }
1376
1377         if (!strcmp(callback_data.name, "pause")) {
1378             tekChainThrow(tekPauseEvent(1));
1379             return SUCCESS;
1380         }
1381
1382         if (!strcmp(callback_data.name, "play")) {
1383             tekChainThrow(tekPauseEvent(0));
1384             return SUCCESS;
1385         }
1386
1387         if (!strcmp(callback_data.name, "step")) {
1388             TekEvent event = {};
1389             event.type = STEP_EVENT;
1390             tekChainThrow(pushEvent(&event_queue, event));
1391             return SUCCESS;
1392         }
1393
1394         // if not a button press, it means the speed or rate has
1395         // been changed.
1396         // need to ensure that the speed+rate is an achievable
1397         // pace for the engine.
1398         // limit to around 100 updates/second
1399         double speed;
1400         tekChainThrow(tekGuiReadNumberOption(window, "speed",
1401 &speed));
1402         if (speed < 0.01)
1403             speed = 0.01;
1404         if (speed > 100)
1405             speed = 100;

```

```

1398
1399     double rate;
1400     tekChainThrow(tekGuiReadNumberOption(window, "rate",
1401 &rate));
1401     if (rate < 1)
1402         rate = 1;
1403     if (rate > 100)
1404         rate = 100;
1405
1406     if (1 / rate / speed > 0.1) {
1407         rate = 10.0 / speed;
1408     }
1409
1410     // push event
1411     tekChainThrow(tekSimulationSpeedEvent(rate, speed,
window));
1412
1413     return SUCCESS;
1414 }
1415
1416 /**
1417 * The window draw callback for the inspect window. Called
every frame that the window is drawn.
1418 * @param window The window / inspect window.
1419 * @throws OPENGL_EXCEPTION .
1420 */
1421 static exception tekInspectDrawCallback(TekGuiWindow* window)
{
1422     // draw the inspect text with slight offset to top left of
the window
1423     TekText* inspect_text = window->data;
1424     tekChainThrow(tekDrawText(inspect_text, (float)(window-
>x_pos + 10), (float)(window->y_pos + 10)));
1425
1426     return SUCCESS;
1427 }
1428
1429
1430 /**
1431 * Write the inspector text, wrapper around a call to sprintf
that has the format string.
1432 * @param string The string to output the inspect text to.

```

```

1433     * @param max_length The maximum allowed length / size of the
string buffer provided.
1434     * @param time The current simulation time.
1435     * @param fps The current fps.
1436     * @param name The name of the object being inspected.
1437     * @param position The position of the object being inspected.
1438     * @param velocity The velocity of the object being inspected.
1439     * @return The number of characters that could not be written
because they did not fit in the buffer.
1440 */
1441 static int tekWriteInspectText(char* string, size_t
max_length, const float time, const float fps, const char* name,
const vec3 position, const vec3 velocity) {
1442     // wrapper around sprintf.
1443     return sprintf(
1444         string, max_length,
1445         "Time: %.3f\nFPS: %.3f\n\nObject Name: %s\nPosition:
(% .5f, %.5f, %.5f)\nVelocity: (%.5f, %.5f, %.5f)\nSpeed: %f\n\nUse
up and down arrows to switch.",
1446         time, fps, name, EXPAND_VEC3(position),
EXPAND_VEC3(velocity), glm_vec3_norm(velocity)
1447     );
1448 }
1449
1450 /**
1451     * Update the inspection text with new information that is
useful to the user.
1452     * @param inspect_text The inspect text mesh in its current
form to be updated.
1453     * @param time The current simulation time in seconds.
1454     * @param fps The current fps to display.
1455     * @param name The name of the object being inspected.
1456     * @param position The position of the inspected object.
1457     * @param velocity The velocity of the inspected object.
1458     * @throws MEMORY_EXCEPTION if malloc() fails.
1459 */
1460 static exception tekUpdateInspectText(TekText* inspect_text,
const float time, const float fps, const char* name, const vec3
position, const vec3 velocity) {
1461     // get lenght of buffer needed to fit inspect text
1462     const int len_buffer = tekWriteInspectText(NULL, 0, time,
fps, name, position, velocity) + 1;
1463

```

```

1464     // alloca is real!
1465     char* buffer = alloca(len_buffer * sizeof(char));
1466     if (!buffer)
1467         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for inspect text.");
1468
1469     // write the inspect buffer
1470     tekWriteInspectText(buffer, len_buffer, time, fps, name,
position, velocity);
1471     buffer[len_buffer - 1] = 0;
1472
1473     // update text with new inspect buffer
1474     tekChainThrow(tekUpdateText(inspect_text, buffer, 16));
1475
1476     // no need to free(buffer) because of alloca
1477     // never knew this, but apparently its hated cuz it
allocates on the stack
1478     return SUCCESS;
1479 }
1480
1481 /**
1482  * Create the runner menu, which has options related to
controlling the running of the scenario such as speed and time
controls.
1483  * @param gui The gui components.
1484  * @throws MEMORY_EXCEPTION if malloc() fails.
1485  * @throws FILE_EXCEPTION if the runner window file does not
exist.
1486 */
1487 static exception tekCreateRunnerMenu(struct TekGuiComponents*
gui) {
1488     // runner window, has options and stuff
1489
tekChainThrow(tekGuiCreateOptionWindow("../res/windows/runner.yml",
&gui->runner_window));
1490     tekChainThrow(tekSimulationSpeedEvent(DEFAULT_RATE,
DEFAULT_SPEED, &gui->runner_window));
1491     gui->runner_window.callback = tekRunnerCallback;
1492
1493     // get font to draw inspector window
1494     TekBitmapFont* font;
1495     tekChainThrow(tekGuiGetDefaultFont(&font));

```

```

1496     tekChainThrow(tekCreateText("", 16, font, &gui->inspect_text));
1497
1498     // inspector window that has text in it
1499     tekChainThrow(tekGuiCreateWindow(&gui->inspect_window));
1500     tekChainThrow(tekGuiSetWindowTitle(&gui->inspect_window,
1501 "Inspect"));
1501     gui->inspect_window.draw_callback =
1502         tekInspectDrawCallback; // <-- manual draw method
1503     gui->inspect_window.data = &gui->inspect_text; // <-- here
1504     is the text in it, but need to manually draw
1505
1506
1507 /**
1508 * Draw the runner menu to the screen.
1509 * @param gui The gui components.
1510 * @throws OPENGL_EXCEPTION if there was a rendering error.
1511 */
1512 static exception tekDrawRunnerMenu(struct TekGuiComponents* gui) {
1513     tekChainThrow(tekGuiDrawAllWindows()); // this will handle
1514     everything cuz we made other windows invisible
1515     return SUCCESS;
1516 }
1517 /**
1518 * Delete the runner memory, freeing any allocated memory for
1519 * this section of the gui.
1520 * @param gui The gui components.
1521 */
1522 static void tekDeleteRunnerMenu(struct TekGuiComponents* gui)
{
1523     // delete everything we allocated
1524     tekGuiDeleteOptionWindow(&gui->runner_window); // run
1525     options
1526     tekGuiDeleteWindow(&gui->inspect_window); // inspector
1527     window
1528     tekDeleteText(&gui->inspect_text); // inspector text
1529 }
1530 /**

```

```

1529     * Choose a random line from the splash.txt file and allocate
a new buffer to store the line in.
1530     * @param buffer A pointer to where the new buffer is to be
allocated.
1531     * @throws MEMORY_EXCEPTION if either of alloca() or malloc()
fails.
1532     * @throws LIST_EXCEPTION if attempted to pick a splash out of
range.
1533     */
1534 static exception tekGetSplashText(char** buffer) {
1535     const char* splash_file = "../res/splash.txt";
1536
1537     uint len_file;
1538     tekChainThrow(getFileSize(splash_file, &len_file));
1539
1540     // i just discovered this "alloca", it automatically frees
after function returns
1541     // so sick
1542     char* file_buffer = alloca(len_file * sizeof(char));
1543     if (!file_buffer)
1544         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
to read splash text file.");
1545
1546     tekChainThrow(readFile(splash_file, len_file,
file_buffer));
1547
1548     // list to store the indices of the start of each line
1549     List line_start_indices = {};
1550     listCreate(&line_start_indices);
1551
1552     // always going to be a line to start with.
1553     tekChainThrow(listAddItem(&line_start_indices, 0));
1554     for (uint i = 0; i < len_file; i++) {
1555         if (file_buffer[i] == '\n')
1556             tekChainThrowThen(listAddItem(&line_start_indices,
(void*)(i + 1)), { listDelete(&line_start_indices); });
1557     }
1558
1559     // add an index at the end of the file
1560     // because to get the line we are going to substring from
current index to next index.
1561     // this will be the next index if the last line is chosen

```

```

1562     tekChainThrowThen(listAddItem(&line_start_indices,
1563     (void*)len_file), { listDelete(&line_start_indices); });
1564     // generate random number
1565     srand(time(0));
1566     const uint list_index = (uint)rand() %
1567     (line_start_indices.length - 1);
1568     // get the line that was randomly picked
1569     void* line_start_index, * line_end_index;
1570     tekChainThrowThen(listGetItem(&line_start_indices,
1571     list_index, &line_start_index),
1572     { listDelete(&line_start_indices); });
1573     tekChainThrowThen(listGetItem(&line_start_indices,
1574     list_index + 1, &line_end_index), { listDelete(&line_start_indices);
1575     });
1576     listDelete(&line_start_indices);
1577
1578     const uint len_line = (uint)line_end_index -
1579     (uint)line_start_index;
1580     // make buffer for splash text.
1581     *buffer = malloc(len_line * sizeof(char));
1582     if (!*buffer)
1583         tekThrow(MEMORY_EXCEPTION, "Failed to allocate buffer
for splash text line.");
1584     memcpy(*buffer, file_buffer + (uint)line_start_index,
1585     len_line);
1586     (*buffer)[len_line - 1] = 0;
1587
1588     return SUCCESS;
1589 }
1590 /**
1591 * Create the menus system. Will call all the sub menu
1592 creation functions.
1593 * @param gui The gui components
1594 * @throws MEMORY_EXCEPTION if malloc() fails.
1595 */
1596 static exception tekCreateMenu(struct TekGuiComponents* gui) {
1597     // create version font + text.
1598     TekBitmapFont* font;
1599     tekChainThrow(tekGuiGetDefaultFont(&font));

```

```

1595     tekChainThrow(tekCreateText("TekPhysics v1.0 Release", 16,
font, &gui->version_text));
1596
1597     // little spinny message on the start screen
1598     char* splash_text;
1599     tekChainThrow(tekGetSplashText(&splash_text));
1600     tekChainThrowThen(tekCreateText(splash_text, 20, font,
&gui->splash_text), { free(splash_text); });
1601     free(splash_text);
1602
1603     // create different sub menus
1604     tekChainThrow(tekCreateMainMenu(WINDOW_WIDTH,
WINDOW_HEIGHT, gui));
1605     tekChainThrow(tekCreateBuilderMenu(WINDOW_WIDTH,
WINDOW_HEIGHT, gui));
1606     tekChainThrow(tekCreateRunnerMenu(gui));
1607
1608     // create save and load windows
1609
tekChainThrow(tekGuiCreateOptionWindow("../res/windows/save.yml",
&gui->save_window));
1610     gui->save_window.callback = tekSaveCallback;
1611
tekChainThrow(tekGuiCreateOptionWindow("../res/windows/load.yml",
&gui->load_window));
1612     gui->load_window.callback = tekLoadCallback;
1613
1614     tekChainThrow(tekSwitchToMainMenu(gui));
1615     return SUCCESS;
1616 }
1617
1618 /**
1619 * Draws the menu. Decides which draw function to call based
on the current mode.
1620 * @param gui The gui components.
1621 * @throws OPENGL_EXCEPTION if there was a rendering error.
1622 */
1623 static exception tekDrawMenu(struct TekGuiComponents* gui) {
1624     int window_width, window_height;
1625     tekGetWindowSize(&window_width, &window_height);
1626
1627     if (next_mode != -1) {
1628         tekChangeMenuMode(gui);

```

```

1629     }
1630
1631     tekSetDrawMode(DRAW_MODE_GUI); // enable gui draw mode so
that windows draw on top of each other
1632
1633     switch (mode) {
1634         case MODE_MAIN_MENU: // differentiate based on what mode
we are to select correct draw function
1635             tekChainThrow(tekDrawMainMenu(gui));
1636             break;
1637         case MODE_RUNNER:
1638             tekChainThrow(tekDrawRunnerMenu(gui));
1639             break;
1640         case MODE_BUILDER:
1641             tekChainThrow(tekDrawBuilderMenu(gui));
1642             break;
1643         case MODE_SAVE:
1644         case MODE_LOAD:
1645             tekGuiDrawAllWindows();
1646             break;
1647         default:
1648             break;
1649     }
1650
1651     // version text is always drawn regardless of mode
1652     tekChainThrow(tekDrawText(&gui->version_text,
(float)window_width - gui->version_text.width - 5.0f,
(float)window_height - gui->version_text.height - 5.0f));
1653
1654     tekSetDrawMode(DRAW_MODE_NORMAL); // disable gui draw mode
so that drawing entities is normal.
1655
1656     return SUCCESS;
1657 }
1658
1659 /**
1660 * Delete the menu system and call the delete functions of the
sub menus.
1661 * @param gui The gui components to delete.
1662 */
1663 static void tekDeleteMenu(struct TekGuiComponents* gui) {
1664     // delete everything we made before
1665     tekDeleteText(&gui->version_text);

```

```

1666     tekDeleteMainMenu(gui);
1667     tekDeleteBuilderMenu(gui);
1668     tekGuiDeleteOptionWindow(&gui->save_window);
1669     tekGuiDeleteOptionWindow(&gui->load_window);
1670     tekDeleteRunnerMenu(gui);
1671 }
1672
1673 /**
1674 * Clean up everything that was allocated for the program.
1675 */
1676 #define tekRunCleanup() \
1677     pushEvent(&event_queue, quit_event); \
1678     tekAwaitEngineStop(engine_thread); \
1679     tekDeleteMenu(&gui); \
1680     tekDeleteScenario(&scenario); \
1681     vectorDelete(&bodies); \
1682     vectorDelete(&entities); \
1683     tekDelete() \
1684
1685 /**
1686 * The actual main function of the program. Sets up the
window, the rendering loop, the gui, the physics engine, and
everything else.
1687 * @throws EXCEPTION Can throw all exceptions for many
different reasons.
1688 */
1689 static exception run() {
1690     // set up GLFW window and other utilities.
1691     tekChainThrow(tekInit("TekPhysics", WINDOW_WIDTH,
WINDOW_HEIGHT));
1692
1693     // set up callbacks for window.
1694     tekChainThrowThen(tekAddKeyCallback(tekMainKeyCallback), {
1695         tekDelete();
1696     });
1697
1698     tekChainThrowThen(tekAddMousePosCallback(tekMainMousePosCallback), {
1699         tekDelete();
1700
1701     tekChainThrowThen(tekAddMouseButtonCallback(tekMainMouseButtonCallback), {
1702         tekDelete();

```

```

1702     });
1703
1704     // create state queue and thread queue, allow to
communicate with thread.
1705     ThreadQueue state_queue = {};
1706     tekChainThrowThen(threadQueueCreate(&event_queue, 4096), {
1707         tekDelete();
1708     });
1709     tekChainThrowThen(threadQueueCreate(&state_queue, 4096), {
1710         threadQueueDelete(&event_queue);
1711         tekDelete();
1712     });
1713
1714     // create vector to store all entities to be drawn.
1715     Vector entities = {};
1716     tekChainThrowThen(vectorCreate(0, sizeof(TekEntity),
&entities), {
1717         threadQueueDelete(&state_queue);
1718         threadQueueDelete(&event_queue);
1719         tekDelete();
1720     });
1721
1722     // create vector to store body snapshots / starting state
of bodies.
1723     Vector bodies = {};
1724     tekChainThrowThen(vectorCreate(0, sizeof(TekBodySnapshot),
&bodies), {
1725         vectorDelete(&entities);
1726         threadQueueDelete(&state_queue);
1727         threadQueueDelete(&event_queue);
1728         tekDelete();
1729     });
1730
1731     // initialise the engine.
1732     TekEvent quit_event;
1733     quit_event.type = QUIT_EVENT;
1734     unsigned long long engine_thread;
1735     tekChainThrowThen(tekInitEngine(&event_queue,
&state_queue, 1.0 / 30.0, &engine_thread), {
1736         vectorDelete(&bodies);
1737         vectorDelete(&entities);
1738         threadQueueDelete(&event_queue);
1739         threadQueueDelete(&state_queue);

```

```

1740         tekDelete();
1741     });
1742
1743     // some random but useful values
1744     vec3 camera_position = {0.0f, 0.0f, 0.0f};
1745     vec3 camera_rotation = {0.0f, 0.0f, 0.0f};
1746
1747     struct TekGuiComponents gui = {};
1748
1749     TekScenario scenario = {};
1750
1751     tekChainThrowThen(tekCreateMenu(
1752         &gui
1753     ), {
1754         tekDeleteScenario(&scenario);
1755         vectorDelete(&bodies);
1756         vectorDelete(&entities);
1757         threadQueueDelete(&event_queue);
1758         threadQueueDelete(&state_queue);
1759         tekDelete();
1760     });
1761
1762     // create the camera struct
1763     TekCamera camera = {};
1764     tekChainThrowThen(tekCreateCamera(&camera,
1765         camera_position, camera_rotation, 1.2f, 0.1f, 100.0f), {
1766         tekRunCleanup();
1767     });
1768
1769     struct timespec curr_time, prev_time;
1770     float delta_time = 0.0f, frame_time = 0.0f, fps = 0.0f;
1771     uint frame_count = 0;
1772     clock_gettime(CLOCK_MONOTONIC, &prev_time);
1773     flag force_exit = 0;
1774     TekState state = {};
1775
1776     // THE infamous main loop
1777     while (tekRunning()) {
1778         // receive all states from the state queue
1779         while (recvState(&state_queue, &state) == SUCCESS) {
1780             switch (state.type) { // time to differentiate
occasions

```

```

1780             case MESSAGE_STATE: // print out a message
received from engine
1781                 if (state.data.message) {
1782                     printf("%s", state.data.message);
1783                     free(state.data.message);
1784                 }
1785                 break;
1786             case EXCEPTION_STATE: // stop the program because
engine had an error
1787                 force_exit = 1;
1788                 tekChainThrowThen(state.data.exception, {
1789                     tekRunCleanup();
1790                     threadQueueDelete(&event_queue);
1791                     threadQueueDelete(&state_queue);
1792                 });
1793                 break;
1794             case ENTITY_CREATE_STATE: // create a new entity
visually
1795                 TekEntity dummy_entity = {};
1796                 // is this a new entity or overwriting a
previously used id?
1797                 // either way, make some space for it
1798                 if (state.object_id == entities.length) {
1799                     tekChainThrowThen(vectorAddItem(&entities,
&dummy_entity), { tekRunCleanup(); });
1800                 } else {
1801                     tekChainThrowThen(vectorSetItem(&entities,
state.object_id, &dummy_entity), { tekRunCleanup(); });
1802                 }
1803
1804                 // create the entity at that point in the
entity list
1805                 TekEntity* create_entity = 0;
1806                 vec3 default_scale = { 1.0f, 1.0f, 1.0f };
1807                 tekChainThrowThen(vectorGetItemPtr(&entities,
state.object_id, &create_entity), { tekRunCleanup(); });
1808
tekChainThrowThen(tekCreateEntity(state.data.entity.mesh_filename,
state.data.entity.material_filename, state.data.entity.position,
state.data.entity.rotation, default_scale, create_entity),
{ tekRunCleanup(); });
1809                 break;

```

```

1810         case ENTITY_UPDATE_STATE: // update an existing
entity
1811             TekEntity* update_entity = 0;
1812             tekChainThrowThen(vectorGetItemPtr(&entities,
state.object_id, &update_entity), { tekRunCleanup(); });
1813             tekUpdateEntity(update_entity,
state.data.entity_update.position,
state.data.entity_update.rotation);
1814             break;
1815         case ENTITY_DELETE_STATE: // delete an entity
1816             TekEntity* delete_entity;
1817             tekChainThrowThen(vectorGetItemPtr(&entities,
state.object_id, &delete_entity), { tekRunCleanup(); });
1818             memset(delete_entity, 0, sizeof(TekEntity));
1819             break;
1820         case INSPECT_STATE: // display some info about an
entity
1821             char* inspect_name;
1822             vec3 position, velocity;
1823             if (inspect_index < 0) {
1824                 inspect_name = "<no body selected>";
1825                 glm_vec3_zero(position);
1826                 glm_vec3_zero(velocity);
1827             } else {
1828
tekChainThrow(tekScenarioGetName(&active_scenario,
(uint)inspect_index, &inspect_name));
1829                 glm_vec3_copy(state.data.inspect.position,
position);
1830                 glm_vec3_copy(state.data.inspect.velocity,
velocity);
1831             }
1832             tekChainThrow(tekUpdateInspectText(
1833                 &gui.inspect_text,
1834                 state.data.inspect.time, fps,
1835                 inspect_name, position, velocity
1836             ));
1837             break;
1838         }
1839
1840         if (force_exit) break;
1841     }
1842

```

```

1843     if (force_exit) break;
1844
1845     // draw entities if in the correct mode
1846     switch (mode) {
1847         case MODE_MAIN_MENU:
1848             break;
1849         case MODE_RUNNER:
1850         case MODE_BUILDER:
1851             for (uint i = 0; i < entities.length; i++) {
1852                 TekEntity* entity;
1853                 tekChainThrowThen(vectorGetItemPtr(&entities,
1854 i, &entity), { tekRunCleanup(); });
1855                 if (entity->mesh == 0) continue;
1856                 tekChainThrowThen(tekDrawEntity(entity,
1857 &camera), { tekRunCleanup(); });
1858             }
1859             break;
1860         default:
1861             break;
1862     }
1863     // draw gui elements.
1864     tekChainThrowThen(tekDrawMenu(
1865         &gui
1866     ), {
1867         tekRunCleanup();
1868     });
1869     tekChainThrow(tekUpdate());
1870
1871     // clock to get delta time, used to make camera and
1872     // movement independent of framerate.
1873     clock_gettime(CLOCK_MONOTONIC, &curr_time);
1874     delta_time = (float)(curr_time.tv_sec -
1875     prev_time.tv_sec) + (float)(curr_time.tv_nsec - prev_time.tv_nsec) /
1876     (float)BILLION;
1877     frame_time += delta_time;
1878
1879     // counting every 10 frames to get an fps readout
1880     if (frame_count == 10) {
1881         frame_count = 0;
1882         fps = 10.0f / frame_time;
1883         frame_time = 0.0f;

```

```

1881         }
1882         frame_count++;
1883
1884         memcpy(&prev_time, &curr_time, sizeof(struct
timespec));
1885
1886         // camera controls, move camera when right clicking.
1887         if (mouse_moved) {
1888             if (mouse_right) {
1889                 camera_rotation[0] = (float)
(fmod(camera_rotation[0] + mouse_dx * delta_time * MOUSE_SENSITIVITY
+ M_PI, 2.0 * M_PI) - M_PI);
1890                 camera_rotation[1] =
(float)fmin(fmax(camera_rotation[1] - mouse_dy * delta_time *
MOUSE_SENSITIVITY, -M_PI_2), M_PI_2);
1891             }
1892             mouse_moved = 0;
1893         }
1894
1895         const double pitch = camera_rotation[1], yaw =
camera_rotation[0];
1896
1897         // movement controls - go towards camera direction.
1898         if (w_pressed) {
1899             camera_position[0] += (float)(cos(yaw) *
cos(pitch) * MOVE_SPEED * delta_time);
1900             camera_position[1] += (float)(sin(pitch) *
MOVE_SPEED * delta_time);
1901             camera_position[2] += (float)(sin(yaw) *
cos(pitch) * MOVE_SPEED * delta_time);
1902         }
1903         if (a_pressed) {
1904             camera_position[0] += (float)(cos(yaw + M_PI_2) *
-MOVE_SPEED * delta_time);
1905             camera_position[2] += (float)(sin(yaw + M_PI_2) *
-MOVE_SPEED * delta_time);
1906         }
1907         if (s_pressed) {
1908             camera_position[0] += (float)(cos(yaw) *
cos(pitch) * -MOVE_SPEED * delta_time);
1909             camera_position[1] += (float)(sin(pitch) * -
MOVE_SPEED * delta_time);

```

```

1910             camera_position[2] += (float)(sin(yaw) *
cos(pitch) * -MOVE_SPEED * delta_time);
1911         }
1912         if (d_pressed) {
1913             camera_position[0] += (float)(cos(yaw + M_PI_2) *
MOVE_SPEED * delta_time);
1914             camera_position[2] += (float)(sin(yaw + M_PI_2) *
MOVE_SPEED * delta_time);
1915         }
1916
1917         // update camera so that we actually are looking where
we want.
1918         tekSetCameraPosition(&camera, camera_position);
1919         tekSetCameraRotation(&camera, camera_rotation);
1920     }
1921
1922     tekRunCleanup();
1923
1924     return SUCCESS;
1925 }
1926
1927 /**
1928  * The entrypoint of the code. Mostly just a wrapper around
the \ref run function.
1929  * @return The exception code, or 0 if there were no
exceptions.
1930 */
1931 int main(void) {
1932     // welcome to tekphysics :D
1933     printf("There was %sa collision.\n", tekTriangleTest() ?
"" : "not ");
1934     return SUCCESS;
1935
1936     tekInitExceptions();
1937     const exception tek_exception = run();
1938     tekLog(tek_exception);
1939     tekCloseExceptions();
1940     return tek_exception;
1941 }
```

tekgl.h

```

1 #pragma once
2
```

```

3  typedef char flag;
4  typedef unsigned char byte;
5  typedef unsigned int uint;
6
7  /**
8   * Take a 3 component vector and expand it, used for printing
mostly.
9  */
10 #define EXPAND_VEC3(vector) vector[0], vector[1], vector[2]
11
12 /**
13  * Take a 4 component vector and expand it, used for printing
mostly.
14 */
15 #define EXPAND_VEC4(vector) vector[0], vector[1], vector[2],
vector[3]
16
17 // different modes that the simulation can be in
18 #define MODE_MAIN_MENU 0
19 #define MODE_BUILDER 1
20 #define MODE_RUNNER 2
21 #define MODE_SAVE 3
22 #define MODE_LOAD 4
23
24 // in case i accidentally type the wrong amount of zeroes.
25 #define BILLION 1000000000

```

core/bitset.c

```

1  #include "bitset.h"
2
3  #include <stdlib.h>
4  #include <string.h>
5
6  #define BITS_PER_INDEX (sizeof(uint64_t) * 8)
7
8  /**
9   * Initialise a BitSet struct by allocating the internal array
that will store the bits.
10  * @note When using a non-growing bitset, the number of bits
available will be rounded to the nearest 64.
11  * @param num_bits The initial number of bits required in the
bitset.

```

```

12  * @param grows Set to 0 if the bitset should not grow past its
original size, 1 if it should be allowed to grow.
13  * @param bitset A pointer to an existing BitSet struct that
will be initialised.
14  * @throws MEMORY_EXCEPTION if malloc() fails.
15  */
16 exception bitsetCreate(uint num_bits, const flag grows, BitSet*
bitset) {
17     // 0 bits will cause a malloc(0) to happen, which ruins
everything ever.
18     if (num_bits == 0)
19         num_bits = 1;
20
21     // ceil division to ensure that there are enough bits.
22     // e.g. sizeof(uint64_t) = 8, so if we had to store 65 bits
for example
23     // normal division would return 65 / 8 = 8. which is one
less number than needed.
24     // (65 + 8 - 1) / 8 -> 72 / 8 = 9 which is what we needed.
25     const uint num_integers = (num_bits + BITS_PER_INDEX - 1) /
BITS_PER_INDEX;
26     bitset->bitset = (uint64_t*)calloc(num_integers,
sizeof(uint64_t));
27     if (!bitset->bitset)
28         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for bitset.");
29
30     bitset->size = num_integers;
31     bitset->grows = grows;
32
33     return SUCCESS;
34 }
35
36 /**
37  * Delete a bitset struct by freeing the internal array, and
zeroing the struct.
38  * @param bitset The bitset to delete.
39  */
40 void bitsetDelete(BitSet* bitset) {
41     // free allocated memory
42     if (bitset->bitset)
43         free(bitset->bitset);
44

```

```

45     // prevent misuse
46     bitset->bitset = 0;
47     bitset->size = 0;
48     bitset->grows = 0;
49 }
50
51 /**
52 * Return the two indices that locate a single bit in the
53 * bitset.
54 * @param bitset_index The index of the bit to retrieve.
55 * @param array_index The index of which number in the array
56 * contains that bit.
57 * @param bit_index The index of the bit (0-63) inside that
58 * number.
59 */
60 static void bitsetGetIndices(const uint bitset_index, uint*
array_index, uint* bit_index) {
61     *array_index = bitset_index / BITS_PER_INDEX; // grows at
62     // 1/64th of the rate of the bitset index
63     *bit_index = bitset_index % BITS_PER_INDEX; // loops 0-63
64     every array index
65 }
66
67 /**
68 * Double the capacity of a bitset, used when space has run
69 * out.
70 * @param bitset The bitset to expand.
71 * @throws MEMORY_EXCEPTION if realloc() fails.
72 */
73 static exception bitsetDoubleSize(BitSet* bitset) {
74     const uint new_size = bitset->size << 1;
75     uint64_t* new_array = (uint64_t*)realloc(bitset->bitset,
76     new_size * sizeof(uint64_t));
77     if (!new_array)
78         tekThrow(MEMORY_EXCEPTION, "Failed to reallocate memory
for bitset.");
79
80     // realloc(...) gives uninitialized memory, which will
81     contain randomly set bits.
82     // we would expect all new bits to be unset, so use
83     memset(...) to clear all bits.
84     // this code below works because we are doubling the size,
85     so the halfway point is where new memory starts e.g. the old length.

```

```

76      // length is now doubled, so the original will be half of
the size, and be the correct length to clear.
77      memset(new_array + bitset->size, 0, bitset->size *
sizeof(uint64_t));
78
79      bitset->bitset = new_array;
80      bitset->size = new_size;
81      return SUCCESS;
82 }
83
84 /**
85  * Helper method to update the value of a bit at a certain
index, and grow array if needed.
86  * @param bitset The bitset to set a value in.
87  * @param index The bit index to set.
88  * @param value The value to set at that index (0 or not 0).
89  * @throws MEMORY_EXCEPTION if the array needed to grow but
couldn't.
90  * @throws BITSET_EXCEPTION if an index out of range was set,
and growth is disabled.
91 */
92 static exception bitsetSetValue(BitSet* bitset, const uint
index, const char value) {
93     uint array_index, bit_index;
94     bitsetGetIndices(index, &array_index, &bit_index);
95
96     // using a loop here, in extreme cases (e.g. setting 1
millionth bit and initialised to size of 1) doubling size may not be
enough to fit the new bit in.
97     // this will repeat until the array is big enough
98     while (array_index >= bitset->size) {
99         if (bitset->grows) {
100             tekChainThrow(bitsetDoubleSize(bitset));
101         } else {
102             tekThrow(BITSET_EXCEPTION, "Attempted to set bit at
a non-existent index.");
103         }
104     }
105
106     // major bit hacking
107     if (value) {
108         bitset->bitset[array_index] |= 1ull << bit_index; //
makes sense

```

```

109     } else {
110         bitset->bitset[array_index] &= ~(1ull << bit_index); // makes sense if u think abt it
111     }
112     return SUCCESS;
113 }
114
115 /**
116  * Set a bit to 1 at the specified index.
117  * @param bitset The bitset to update the bit of.
118  * @param index The index of the bit to set.
119  * @throws MEMORY_EXCEPTION if the bitset tried to grow but failed.
120  * @throws BITSET_EXCEPTION if the index is out of range, and growth is disabled.
121 */
122 exception bitsetSet(BitSet* bitset, const uint index) {
123     // set = set to 1
124     tekChainThrow(bitsetSetValue(bitset, index, 1));
125     return SUCCESS;
126 }
127
128 /**
129  * Set a bit to 0 at the specified index.
130  * @param bitset The bitset to update the bit of.
131  * @param index The index of the bit to unset.
132  * @throws MEMORY_EXCEPTION if the bitset tried to grow but failed.
133  * @throws BITSET_EXCEPTION if the index is out of range, and growth is disabled.
134 */
135 exception bitsetUnset(BitSet* bitset, const uint index) {
136     // unset = set to 0
137     tekChainThrow(bitsetSetValue(bitset, index, 0));
138     return SUCCESS;
139 }
140
141 /**
142  * Get the value of the bit at a specific index.
143  * @param bitset The bitset to retrieve from.
144  * @param index The index of the bit to retrieve.
145  * @param value A pointer to where the value of the bit (0 or 1) will be stored.

```

```

146  * @throws BITSET_EXCEPTION if the index is out of range, and
growth is disabled.
147  */
148 exception bitsetGet(const BitSet* bitset, const uint index,
flag* value) {
149     // need to access in two stages, array index and bit index
150     // array has 64 bit integers in it. only want one of those
bits
151     uint array_index, bit_index;
152     bitsetGetIndices(index, &array_index, &bit_index);
153
154     if (array_index >= bitset->size) {
155         if (bitset->grows)
156             // if out of range, bit is unset. must be 0.
157             return 0;
158         tekThrow(BITSET_EXCEPTION, "Could not access bit
outside of range.");
159     }
160
161     // shift by bit index to get actual bit value.
162     *value = (bitset->bitset[array_index] & (1ull <<
bit_index)) ? 1 : 0;
163     return SUCCESS;
164 }
165
166 /**
167  * Helper function to get a 1D index from a 2D coordinate.
168  * The index snakes across a grid, so if the bitset expands,
the coordinates still map to the same index in the array.
169  * All credit goes to me at 2:30am on a random saturday for
this idea.
170  * @param x The x coordinate.
171  * @param y The y coordinate.
172  * @return The 1D index.
173 */
174 static uint bitsetGet1DIndex(const uint x, const uint y) {
175     // 0x5F3759DF evil floating point bit level hacking
176     // yea i dont remember how it works exactly but u can
figure it out
177     if (y > x) {
178         return y * y + 2 * y - x;
179     }
180     return x * x + y;

```

```

181 }
182 /**
183 * Set the bitset at a coordinate (x, y)
184 * @param bitset The bitset to set
185 * @param x The x coordinate
186 * @param y The y coordinate
187 * @throws MEMORY_EXCEPTION if malloc() fails
188 * * @throws BITSET_EXCEPTION if index out of range, and
189 growth disabled.
190 */
191 exception bitsetSet2D(BitSet* bitset, const uint x, const uint
y) {
192     // wrapper around normal function, just with converting to
1d coordinate.
193     tekChainThrow(bitsetSet(bitset, bitsetGet1DIndex(x, y)));
194     return SUCCESS;
195 }
196
197 /**
198 * Unset the bitset at a coordinate (x, y)
199 * @param bitset The bitset to unset
200 * @param x The x coordinate
201 * @param y The y coordinate
202 * @throws MEMORY_EXCEPTION if malloc() fails
203 * * @throws BITSET_EXCEPTION if index out of range, and
growth disabled.
204 */
205 exception bitsetUnset2D(BitSet* bitset, const uint x, const
uint y) {
206     // wrapper around normal function, just converting
coordinate
207     tekChainThrow(bitsetUnset(bitset, bitsetGet1DIndex(x, y)));
208     return SUCCESS;
209 }
210
211 /**
212 * Get the bitset at a coordinate (x, y)
213 * @param bitset The bitset to set
214 * @param x The x coordinate
215 * @param y The y coordinate
216 * @param value The value of the bitset at that coordinate.

```

```

217  * @throws BITSET_EXCEPTION if index out of range, and growth
disabled.
218  */
219 exception bitsetGet2D(const BitSet* bitset, const uint x, const
uint y, flag* value) {
220     // wrapper around normal bitset function
221     // but get the 1d index
222     tekChainThrow(bitsetGet(bitset, bitsetGet1DIndex(x, y),
value));
223     return SUCCESS;
224 }
225
226 /**
227 * Set all bits to zero.
228 * @param bitset The bitset to operate on.
229 */
230 void bitsetClear(const BitSet* bitset) {
231     // set all bits to zero in chunks of 64 bits
232     memset(bitset->bitset, 0, bitset->size * sizeof(uint64_t));
233 }
```

core/bitset.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "exception.h"
5
6 #include <stdint.h>
7
8 /// BitSet struct, should be initialised with \ref bitsetCreate
before use.
9 typedef struct BitSet {
10     uint size; // The size of the internal array.
11     flag grows; // A flag (0 or 1) that determines whether the
internal array should be able to resize.
12     uint64_t* bitset; // The internal array storing the bits.
13 } BitSet;
14
15 exception bitsetCreate(uint num_bits, flag grows, BitSet*
bitset);
16 void bitsetDelete(BitSet* bitset);
17 exception bitsetSet(BitSet* bitset, uint index);
18 exception bitsetUnset(BitSet* bitset, uint index);
```

```

19 exception bitsetGet(const BitSet* bitset, uint index, flag*
value);
20
21 exception bitsetSet2D(BitSet* bitset, uint x, uint y);
22 exception bitsetUnset2D(BitSet* bitset, uint x, uint y);
23 exception bitsetGet2D(const BitSet* bitset, uint x, uint y,
flag* value);
24
25 void bitsetClear(const BitSet* bitset);

```

core/exception.c

```

1 #include "exception.h"
2
3 #include <stdlib.h>
4 #include <string.h>
5 #include <stdio.h>
6 #include "../tekgl.h"
7
8 // using fixed size arrays, basically built to not have
exceptions because if an exception occurs, there is nothing in place
to handle it yet.
9
10 flag initialised = 0;
11 char exception_buffer[E_BUFFER_SIZE]; // stores the actual
exception that occurred e.g. "Memory Exception in something.c"
12 char stack_trace_buffer[STACK_TRACE_BUFFER_SIZE]
[E_MESSAGE_SIZE]; // stores stack trace e.g. "... in line 21..."
13 uint stack_trace_index = 0;
14 char* default_exception = "Unknown Exception";
15 char* exceptions[NUM_EXCEPTIONS]; // stores name of each
exception e.g. "Memory Exception"
16
17 /**
18 * Add a new exception to the list of named exceptions.
19 * @param exception_code The exception code, via one of the
exception macros.
20 * @param exception_name The human-readable name of the
exception.
21 */
22 void tekAddException(const int exception_code, const char*
exception_name) {
23     const size_t len_exception_name = strlen(exception_name) +
1; // length of string + null terminator

```

```

24     char* buffer = (char*)malloc(len_exception_name);
25     if (!buffer) return; // cant throw an exception cuz no
exception handler yet... oopsies
26     memcpy(buffer, exception_name, len_exception_name);
27     exceptions[exception_code - 1] = buffer;
28 }
29
30 /**
31  * Initialise exceptions, adding human readable names to the
list of exception names.
32 */
33 void tekInitExceptions() {
34     // in no particular order (in the order i realised that
that thing could fail) (drumroll please)
35     // default exception
36     tekAddException(FAILURE, "Failure");
37
38     // C issues
39     tekAddException(MEMORY_EXCEPTION, "Memory Allocation
Exception");
40     tekAddException(NULL_PTR_EXCEPTION, "Null Pointer
Exception");
41
42     // graphics + os stuff
43     tekAddException(GLFW_EXCEPTION, "GLFW Exception");
44     tekAddException(GLAD_EXCEPTION, "GLAD Exception");
45     tekAddException(FILE_EXCEPTION, "File Exception");
46     tekAddException(OPENGL_EXCEPTION, "OpenGL Exception");
47     tekAddException(STBI_EXCEPTION, "STBI Exception");
48     tekAddException(FREETYPE_EXCEPTION, "FreeType Exception");
49
50     // data structures + my code
51     tekAddException(LIST_EXCEPTION, "List Exception");
52     tekAddException(YML_EXCEPTION, "YML Exception");
53     tekAddException(STACK_EXCEPTION, "Stack Exception");
54     tekAddException(HASHTABLE_EXCEPTION, "Hash Table
Exception");
55     tekAddException(QUEUE_EXCEPTION, "Queue Exception");
56     tekAddException(THREAD_EXCEPTION, "Thread Exception");
57     tekAddException(VECTOR_EXCEPTION, "Vector Exception");
58     tekAddException(ENGINE_EXCEPTION, "Engine Exception");
59     tekAddException(BITSET_EXCEPTION, "BitSet Exception");
60

```

```

61     // unit testing
62     tekAddException(ASSERT_EXCEPTION, "Assertion Exception");
63     initialised = 1;
64 }
65
66 /**
67 * Frees all the names of the exceptions that were created.
68 */
69 void tekCloseExceptions() {
70     // exception names are dynamically allocated for some
reason idk
71     for (int i = 0; i < NUM_EXCEPTIONS; i++) {
72         if (exceptions[i]) free(exceptions[i]);
73     }
74     initialised = 0;
75 }
76
77 /**
78 * Get the name of an exception using its numeric code.
79 * @param exception_code The exception code to get.
80 * @return A pointer to the exception's name
81 * @note Will return a default name if the exception code is
not valid, or the exception handler is not initialised.
82 */
83 char* tekGetExceptionName(const int exception_code) {
84     if (!initialised || exception_code > NUM_EXCEPTIONS ||
exception_code <= 0)
85         return default_exception;
86     char* exception = exceptions[exception_code - 1];
87     if (!exception) exception = default_exception;
88     return exception;
89 }
90
91 /**
92 * Print out the last exception that occurred, including stack
trace.
93 */
94 void tekPrintException() {
95     if (exception_buffer[0])
96         printf("%s", exception_buffer);
97     else
98         printf("Unknown exception!\n"); // had some issues
before where i tekChainThrow'ed some non-exception-returning

```

functions and it tried to print an exception that wasn't there. This makes it more clear that it happened.

```
99     // print stack trace
100    for (uint i = 0; i < stack_trace_index; i++) {
101        printf("%s", stack_trace_buffer[i]);
102    }
103 }
104
105 /**
106 * Record that an exception has occurred, based on line number,
function and file.
107 * @note Should be used mostly with the 'tekThrow(...)' macro.
108 * @param exception_code The exception code via one of the
macros, e.g. MEMORY_EXCEPTION
109 * @param exception_line The line number of the exception, via
the __LINE__ macro.
110 * @param exception_function The function in which the
exception occurred, should use the __FUNCTION__ macro.
111 * @param exception_file The file where the exception occurred,
should use the __FILE__ macro.
112 * @param exception_message A message to describe the error,
e.g. malloc() returned a null pointer.
113 */
114 void tekSetException(const int exception_code, const int
exception_line, const char* exception_function, const char*
exception_file, const char* exception_message) {
115     // write into the error buffer up to the maximum size of
the buffer.
116     snprintf(exception_buffer, E_BUFFER_SIZE, "%s (%d) in
function '%s', line %d of %s:\n      %s\n",
tekGetExceptionName(exception_code), exception_code,
exception_function, exception_line, exception_file,
exception_message);
117     // if message too long, not space for nullptr at end, so
possibly add it
118     if (exception_buffer[E_BUFFER_SIZE - 2] != 0) {
119         exception_buffer[E_BUFFER_SIZE - 2] = '\n';
120         exception_buffer[E_BUFFER_SIZE - 1] = 0;
121     }
122
123     // restart stack trace buffer
124     stack_trace_index = 0;
125 }
```

```

126
127  /**
128   * Function used to add to the stack trace of an exception.
129   * @note Should be used only with the 'tekChainThrow(...)'
macro.
130   * @param exception_code The exception code, e.g.
MEMORY_EXCEPTION
131   * @param exception_line The line number of the exception, via
the __LINE__ macro.
132   * @param exception_function The function in which the
exception occurred, should use the __FUNCTION__ macro.
133   * @param exception_file The file where the exception occurred,
should use the __FILE__ macro.
134 */
135 void tekTraceException(const int exception_code, const int
exception_line, const char* exception_function, const char*
exception_file) {
136     // Write out the exception as long as possible in the
message buffer.
137     snprintf(stack_trace_buffer[stack_trace_index],
E_MESSAGE_SIZE, "... %s (%d) in function '%s', line %d of %s\n",
tekGetExceptionName(exception_code), exception_code,
exception_function, exception_line, exception_file);
138     // if we blew right past the end of the buffer, the last
char will not be null.
139     // so just chop off at the end
140     if (stack_trace_buffer[stack_trace_index][E_MESSAGE_SIZE -
2] != 0) {
141         stack_trace_buffer[stack_trace_index][E_MESSAGE_SIZE -
2] = '\n';
142         stack_trace_buffer[stack_trace_index][E_MESSAGE_SIZE -
1] = 0;
143     }
144
145     // if the stack trace buffer is full, then just have to
ignore this one to show more important stack trace
146     if (stack_trace_index == STACK_TRACE_BUFFER_SIZE - 1) {
147         snprintf(stack_trace_buffer[stack_trace_index - 1],
E_MESSAGE_SIZE, "... stack trace too large to display entirely ...\\
n");
148     } else {
149         stack_trace_index++;
150     }

```

```
151 }
```

core/exception.h

```
1 #pragma once
2
3 #define E_MESSAGE_SIZE          192
4 #define E_BUFFER_SIZE           128 + E_MESSAGE_SIZE
5 #define STACK_TRACE_BUFFER_SIZE 16
6
7 #define SUCCESS                 0
8 #define FAILURE                 1
9 #define MEMORY_EXCEPTION        2
10 #define NULL_PTR_EXCEPTION     3
11 #define GLFW_EXCEPTION          4
12 #define GLAD_EXCEPTION          5
13 #define FILE_EXCEPTION          6
14 #define OPENGL_EXCEPTION        7
15 #define STBI_EXCEPTION          8
16 #define FREETYPE_EXCEPTION      9
17 #define LIST_EXCEPTION          10
18 #define YML_EXCEPTION           11
19 #define STACK_EXCEPTION         12
20 #define HASHTABLE_EXCEPTION     13
21 #define QUEUE_EXCEPTION         14
22 #define THREAD_EXCEPTION        15
23 #define VECTOR_EXCEPTION        16
24 #define ENGINE_EXCEPTION        17
25 #define BITSET_EXCEPTION        18
26 #define ASSERT_EXCEPTION        19
27 #define NUM_EXCEPTIONS          19
28
29 typedef int exception;
30
31 #define tekExcept(exception_code, exception_message) { const
exception __tek_exception = exception_code; if (__tek_exception)
{ tekSetException(__tek_exception, __LINE__, __FUNCTION__, __FILE__,
exception_message); } }
32 #define tekChainExcept(exception_code) { const exception
__tek_exception = exception_code; if (__tek_exception)
{ tekTraceException(__tek_exception, __LINE__, __FUNCTION__,
__FILE__); } }
33 #define tekChainBreak(exception_code) { const exception
__tek_exception = exception_code; if (__tek_exception)
```

```

{ tekTraceException(__tek_exception, __LINE__, __FUNCTION__,
__FILE__); break; } }

34 #define tekThrow(exception_code, exception_message) { const
exception __tek_exception = exception_code; if (__tek_exception)
{ tekSetException(__tek_exception, __LINE__, __FUNCTION__, __FILE__,
exception_message); return __tek_exception; } }

35 #define tekChainThrow(exception_code) { const exception
__tek_exception = exception_code; if (__tek_exception)
{ tekTraceException(__tek_exception, __LINE__, __FUNCTION__,
__FILE__); return __tek_exception; } }

36 #define tekLog(exception_code) { const exception __tek_exception
= exception_code; if (__tek_exception)
{ tekTraceException(__tek_exception, __LINE__, __FUNCTION__,
__FILE__); tekPrintException(); } }

37

38 #define tekThrowThen(exception_code, exception_message, then)
{ const exception __tek_exception = exception_code; if
(__tek_exception) { tekSetException(__tek_exception, __LINE__,
__FUNCTION__, __FILE__, exception_message); then; return
__tek_exception; } }

39 #define tekChainThrowThen(exception_code, then) { const
exception __tek_exception = exception_code; if (__tek_exception)
{ tekTraceException(__tek_exception, __LINE__, __FUNCTION__,
__FILE__); then; return __tek_exception; } }

40

41 void tekInitExceptions();
42 void tekCloseExceptions();
43 const char* tekGetException();
44 void tekPrintException();
45 void tekSetException(int exception_code, int exception_line,
const char* exception_function, const char* exception_file, const
char* exception_message);
46 void tekTraceException(int exception_code, int exception_line,
const char* exception_function, const char* exception_file);

```

core/file.c

```

1 #include "file.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/stat.h>
7

```

```

8  /**
9   * Return the size of a file in bytes.
10  * @param filename The path that leads to the file.
11  * @param file_size A pointer to a uint where the size will be
recorded.
12  * @throws FILE_EXCEPTION if the file does not exist.
13  */
14 exception getFileSize(const char* filename, uint* file_size) {
15     // linux allows file size to be checked
16     struct stat file_stat;
17     if (stat(filename, &file_stat) == -1)
tekThrow(FILE_EXCEPTION, "Could not read file stat.");
18
19     // file size includes the EOF character, so this already
includes room for a null at the end.
20     *file_size = file_stat.st_size + 1;
21
22     return SUCCESS;
23 }
24
25 /**
26  * Read a file into an existing buffer.
27  * @note Buffer is required to be large enough for the file.
Use the 'getFileSize(...)' function to find the size, and then
allocate memory.
28  * @param filename The path that leads to the file to be read.
29  * @param buffer_size The size of the buffer passed in.
30  * @param buffer The buffer that has been allocated that will
store the contents of the file.
31  * @throws FILE_EXCEPTION if the file does not exist, or is
larger than what can be stored in the buffer.
32 */
33 exception readFile(const char* filename, const uint
buffer_size, char* buffer) {
34     // get file pointer to the file
35     FILE* file_ptr = fopen(filename, "r");
36     if (!file_ptr) tekThrow(FILE_EXCEPTION, "Could not open
file.");
37
38     // iterate over each character until end of file is reached
39     int c; uint count = 0;
40     while ((c = fgetc(file_ptr)) != EOF) {
41         buffer[count++] = (char)c;

```

```

42         if (count == buffer_size) {
43             fclose(file_ptr);
44             tekThrow(FILE_EXCEPTION, "File contents larger than
buffer.");
45         }
46     }
47
48     // add terminating character and close file
49     buffer[count] = 0;
50     fclose(file_ptr);
51
52     return SUCCESS;
53 }
54
55 /**
56  * Write a string buffer into a file. This will create a file
if one doesn't exist, and will overwrite if it does.
57  * @param buffer The contents to write to the file.
58  * @param filename The path to the file that should be written
to.
59  * @throws FILE_EXCEPTION
60  */
61 exception writeFile(const char* buffer, const char* filename) {
62     // get pointer to file.
63     FILE* file_ptr = fopen(filename, "w");
64     if (!file_ptr) tekThrow(FILE_EXCEPTION, "Could not open
file.");
65
66     // write file
67     const int fpf_out = fprintf(file_ptr, "%s", buffer);
68     fclose(file_ptr);
69     if (fpf_out < 0)
70         tekThrow(FILE_EXCEPTION, "Failed to write to file.");
71
72     return SUCCESS;
73 }
74
75 /**
76  * Combine a directory and a file name into a single string.
For example, directory="/usr/tekphys/" filename="file.txt" ->
result="/usr/tekphys/file.txt"
77  * @param[in] directory The directory to start with
78  * @param[in] filename The file name to finish with

```

```

79     * @param[out] result A pointer to a char pointer which will be
overwritten to point to the new buffer.
80     * @throws MEMORY_EXCEPTION if malloc() fails.
81     */
82 exception addPathToFile(const char* directory, const char*
filename, char** result) {
83     // get some lengths
84     const uint len_directory = strlen(directory);
85     const uint len_filename = strlen(filename);
86
87     // final string should only have one null terminator, if it
had one from each string it would cut half way
88     const uint len_result = len_directory + len_filename + 1;
89     *result = (char*)malloc(len_result * sizeof(char)); //  

mallashib
90     if (!*result)
91         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for result.");
92
93     // Python:    result = "directory" + "filename" 👍
94     // TekPhysics:
95     memcpy(*result, directory, len_directory);
96     memcpy(*result + len_directory, filename, len_filename);
97     (*result)[len_directory + len_filename] = 0;
98     // We choose to write in C in this NEA and do the other
things, not because they are easy, but because they are hard;
99     // because that goal will serve to organize and measure the
best of our energies and skills,
100    // because that challenge is one that we are willing to
accept, one we are unwilling to postpone,
101    // and one we intend to win, and the others, too.
102    return SUCCESS;
103 }
104
105 /**
106  * Check if a file exists.
107  * @param filename The file to check.
108  * @return 0 if it does not exist, 1 if it does.
109 */
110 flag fileExists(const char* filename) {
111     FILE* file_ptr = fopen(filename, "r"); // open in read-only
112     if (file_ptr) { // if possible to read, it exists
113         fclose(file_ptr);

```

```

114         return 1;
115     }
116     return 0; // otherwise it effectively does not exist.
117 }
```

core/file.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "exception.h"
5
6 exception getFileSize(const char* filename, uint* file_size);
7 exception readFile(const char* filename, uint buffer_size, char*
buffer);
8 exception writeFile(const char* buffer, const char* filename);
9 exception addPathToFile(const char* directory, const char*
filename, char** result);
10 flag fileExists(const char* filename);
```

core/hashtable.c

```

1 #include "hashtable.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 /**
8  * @brief Delete a hashtable, freeing all the memory that the
hashtable functions allocated.
9  * @note This does not free any user allocated memory, for
example pointers that are stored in the hashtable.
10 * @param[in] hashtable A pointer to the hashtable to delete.
11 */
12 void hashtableDelete(const HashTable* hashtable) {
13     if (!hashtable) return;
14
15     // iterate over every possible index in the hash table
16     for (int i = 0; i < hashtable->length; i++) {
17         HashNode* node_ptr = hashtable->internal[i];
18         // loop through the associated linked list
19         while (node_ptr) {
20             // free the key as we allocated a copy
21             if (node_ptr->key) free(node_ptr->key);
```

```

22          // keep track of pointer so that we can free it
without issues
23          HashNode* temp_ptr = node_ptr->next;
24          free(node_ptr);
25          node_ptr = temp_ptr;
26      }
27  }
28  // finally free the hashtable struct
29  free(hashtable->internal);
30 }
31
32 /**
33 * @brief Create a hashtable. The length is relatively
unimportant, setting it too small just means more rehashes.
34 * @param[out] hashtable A pointer to an empty hashtable
struct.
35 * @param[in] length The starting size of the hashtable. If
this size is filled by 75%, the hashtable will double its size.
36 * @throws MEMORY_EXCEPTION if malloc() fails.
37 */
38 exception hashtableCreate(HashTable* hashtable, const unsigned
int length) {
39     // check for random scenarios such as no hashtable or
already allocated hashtable
40     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot create
hashtable from null ptr.");
41     //if (hashtable->internal) hashtableDelete(hashtable);
42
43     // allocate room for the internal array, and set some
default values
44     hashtable->internal = (HashNode**)calloc(length,
sizeof(HashNode*));
45     if (!hashtable->internal) tekThrow(MEMORY_EXCEPTION,
"Failed to allocate memory for hashtable.");
46     hashtable->length = length;
47     hashtable->num_items = 0;
48     return SUCCESS;
49 }
50
51 /**
52 * @brief Create a hash index for a hashtable given a string.
Essentially works by summing ASCII values of each character, modulo
length of internal array.

```

```

53  * @param[in] hashtable A pointer to the hashtable.
54  * @param[in] key The string key to hash with.
55  * @param[out] hash The returned hash value.
56  * @throws HASHTABLE_EXCEPTION if the hashtable is empty.
57  */
58 static exception hashtableHash(HashTable* hashtable, const
char* key, unsigned int* hash) {
59     // make sure we dont have bogus data
60     if (!hashtable || !key || !hash)
tekThrow(NULL_PTR_EXCEPTION, "Cannot hash key from null ptr.");
61     if (!hashtable->length) tekThrow(HASHTABLE_EXCEPTION,
"Cannot get hash index for an empty hashtable.");
62     // simple hashing algortithm for strings
63     // sum ascii values, and then modulo to stop it exceeding
the maximum size of the internal array
64     const size_t len_key = strlen(key);
65     *hash = 0;
66     for (size_t i = 0; i < len_key; i++) {
67         *hash = (*hash + key[i]) % hashtable->length;
68     }
69     return SUCCESS;
70 }
71
72 /**
73 * @brief Create a node for a hashtable, and insert it into the
hashtable correctly.
74 * @param[in] hashtable A pointer to the hashtable.
75 * @param[in] key The key which names the node.
76 * @param[in] hash The hash index at which to store the node.
77 * @param[out] out_ptr A pointer to store a reference to the
created node for further use.
78 * @throws MEMORY_EXCEPTION if malloc() fails.
79 */
80 static exception hashtableCreateNode(const HashTable*
hashtable, const char* key, const unsigned int hash, HashNode** out_ptr) {
81     // ensure there isn't a null ptr
82     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot create
node from null ptr.");
83
84     // make a temporary node
85     HashNode node = {};
86

```

```

87     // set next ptr to first item of internal array
88     node.next = hashtable->internal[hash];
89
90     // if node.next is null, then it means it is the first item
of the linked list
91     const int first = !node.next;
92
93     // iterate over the list to find the final item
94     HashNode* node_ptr = &node;
95     while (node_ptr->next) node_ptr = node_ptr->next;
96     node_ptr->next = (HashNode*)calloc(1, sizeof(HashNode));
97     if (!node_ptr->next) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for new node.");
98
99     // mallocate some memory to store a copy of the key
100    char* key_copy = (char*)malloc(strlen(key) + 1);
101    if (!key_copy) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for copy of key.");
102    memcpy(key_copy, key, strlen(key) + 1);
103    node_ptr->next->key = key_copy;
104
105    // if this is the first item, update the actual array to
indicate that there is a list here
106    if (first) hashtable->internal[hash] = node_ptr->next;
107
108    // return reference to the new node
109    *out_ptr = node_ptr->next;
110    return SUCCESS;
111 }
112
113 /**
114  * @brief Check if a node exists in the hashtable, and if so
return a pointer to this node.
115  * @note The retrieved node is not a copy, and will be freed
once the hashtable is freed.
116  * @param[in] hashtable A pointer to the hashtable.
117  * @param[in] key The string key at which the node is stored.
118  * @param[out] node_ptr A pointer to store a pointer to the
node if found.
119  * @throws NULL_PTR_EXCEPTION if hashtable is NULL
120  * @returns SUCCESS if the node was found, FAILURE if it was
not.
121 */

```

```

122 static exception hashtableGetNode(HashTable* hashtable, const
123     char* key, HashNode** node_ptr) {
124     // ensure not a null ptr
125     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot get
node from null ptr.");
126
127     // find the hash index
128     unsigned int hash;
129     tekChainThrow(hashtableHash(hashtable, key, &hash));
130
131     // iterate over the hashtable
132     *node_ptr = hashtable->internal[hash];
133     int found = 0;
134     while (*node_ptr) {
135         // if the key is equal to what we are searching for,
then stop
136         if (!strcmp(key, (*node_ptr)->key)) {
137             found = 1;
138             break;
139             *node_ptr = (*node_ptr)->next;
140         }
141
142         // if node pointer was not found, make sure we output this
143         if (!found) *node_ptr = 0;
144         return (found) ? SUCCESS : FAILURE;
145     }
146
147 /**
148 * @brief Return a list of keys that make up the hashtable.
149 * @note The function will allocate an array at the pointer
specified that needs to be freed. The length of this array will be
hashtable.num_items. (hashtable.length specifies the length of the
internal array).
150 * @param hashtable A pointer to the hashtable.
151 * @param keys A pointer to an array of char arrays that will
be allocated and filled with keys.
152 * @throws MEMORY_EXCEPTION if malloc() fails.
153 */
154 exception hashtableGetKeys(const HashTable* hashtable, char***
keys) {
155     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot get
keys from null ptr.");

```

```

156
157     // mallocate some memory to store a list of keys
158     *keys = (char**)malloc(hashtable->num_items *
159     sizeof(char*));
160
161     // iterate over the internal array
162     unsigned int count = 0;
163     for (unsigned int i = 0; i < hashtable->length; i++) {
164         // iterate over each linked list in the array
165         const HashNode* node_ptr = hashtable->internal[i];
166         while (node_ptr) {
167             // add key to the list
168             (*keys)[count++] = node_ptr->key;
169             node_ptr = node_ptr->next;
170         }
171     }
172     return SUCCESS;
173 }
174
175 /**
176  * @brief Return a list of values stored in the hashtable.
177  * @note The function will allocate memory to store the values
178  * in, which needs to be freed. The order of the values will match the
179  * order of keys retrieved by hashtableGetKeys.
180  * @param hashtable A pointer to the hashtable.
181  * @param values A pointer to an array of void pointers that
182  * will be filled with the values contained in the hashtable.
183  * @throws MEMORY_EXCEPTION if could not allocate array to
184  * store values.
185  */
186 exception hashtableGetValues(const HashTable* hashtable,
187     void*** values) {
188     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot get
189     values from null ptr.");
190
191     // allocate some memory to store a list of all values
192     *values = (void**)malloc(hashtable->num_items *
193     sizeof(void*));
194     if (!(*values)) tekThrow(MEMORY_EXCEPTION, "Failed to
195     allocate memory for list of values.");
196
197 }
```

```

189     // iterate over each item in the array
190     unsigned int count = 0;
191     for (unsigned int i = 0; i < hashtable->length; i++) {
192         // iterate over each value in the linked list
193         const HashNode* node_ptr = hashtable->internal[i];
194         while (node_ptr) {
195             // add to list
196             (*values)[count++] = node_ptr->data;
197             node_ptr = node_ptr->next;
198         }
199     }
200     return SUCCESS;
201 }
202
203 /**
204 * @brief Resize the hashtable to be a new size, copying the
205 items into new locations.
206 * @note By the nature of how hashtables work, the order of
207 items from hashtableGetKeys/Values will change after rehashing.
208 * @param hashtable A pointer to the hashtable.
209 * @param new_length The new length required by the hashtable.
210 * @throws MEMORY_EXCEPTION if malloc() fails.
211 */
212 static exception hashtableRehash(HashTable* hashtable, const
213 unsigned int new_length) {
214     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot rehash
from null ptr.");
215
216     // make room to copy the keys and values of the hash table
217     char** keys = (char**)calloc(hashtable->num_items,
218     sizeof(char*));
219     if (!keys) tekThrow(MEMORY_EXCEPTION, "Failed to allocate
memory for copy of keys.");
220     void** values = (void**)calloc(hashtable->num_items,
221     sizeof(void*));
222     if (!values) {
223         free(keys);
224         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for copy of values.");
225     }
226
227     // iterate over each item of the array
228     unsigned int count = 0;

```

```

224     for (unsigned int i = 0; i < hashtable->length; i++) {
225         // iterate over each item in the linked list
226         const HashNode* node_ptr = hashtable->internal[i];
227         while (node_ptr) {
228             // find the length of the key and copy it to a new
array, and store the pointer
229             // also store the value in the values array
230             const size_t len_key = strlen(node_ptr->key) + 1;
231             char* key_copy = (char*)malloc(len_key *
sizeof(char));
232             if (!key_copy) {
233                 for (unsigned int j = 0; j < count; j++)
free(keys[j]);
234                 free(keys);
235                 free(values);
236                 tekThrow(MEMORY_EXCEPTION, "Failed to allocate
memory for copy of key.");
237             }
238             memcpy(key_copy, node_ptr->key, len_key);
239             keys[count] = key_copy;
240             values[count] = node_ptr->data;
241             count++;
242             node_ptr = node_ptr->next;
243         }
244     }
245
246     // create a new hash table with a different size
247     tekChainThrow(hashtableCreate(hashtable, new_length));
248
249     // iterate over each item in the internal array
250     for (unsigned int i = 0; i < count; i++) {
251         // iterate over each item in the linked list
252         // add the record we copied
253         const exception tek_exception = hashtableSet(hashtable,
keys[i], values[i]);
254
255         // catch exception if it occurs
256         if (tek_exception) {
257             for (unsigned int j = 0; j < count; j++)
free(keys[j]);
258             free(keys);
259             free(values);
260             tekChainThrow(tek_exception);

```

```

261         }
262     }
263
264     // free copies of keys
265     for (unsigned int i = 0; i < count; i++) free(keys[i]);
266
267     // free arrays
268     free(keys);
269     free(values);
270     return SUCCESS;
271 }
272
273 /**
274 * @brief Returns whether a hashtable is more than 75% full, at
which point the number of collisions makes the hashtable slower,
requiring the linked lists to be traversed more often.
275 * @param hashtable A pointer to the hashtable.
276 * @return 1 if the internal array is more than 75% full, 0
otherwise
277 */
278 static flag hashtableTooFull(const HashTable* hashtable) {
279     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot check
if null ptr is full.");
280     // return true if the hashtable is more than 75% full
281     return (flag)((4 * hashtable->num_items) >= (3 * hashtable-
>length));
282 }
283
284 /**
285 * Get an item from a hashtable by key.
286 * @param hashtable A pointer to the hashtable.
287 * @param key The key to the wanted data.
288 * @param data A pointer that will be filled with the stored
data.
289 * @throws HASHTABLE_EXCEPTION if the hashtable is empty.
290 */
291 exception hashtableGet(HashTable* hashtable, const char* key,
void** data) {
292     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot get
from null ptr.");
293
294     // find the index of the internal array
295     unsigned int hash;

```

```

296     tekChainThrow(hashtableHash(hashtable, key, &hash));
297
298     // retrieve the node in the linked list
299     HashNode* node_ptr;
300     exception tek_exception = hashtableGetNode(hashtable, key,
301 &node_ptr);
302
303     if (tek_exception) {
304         if (tek_exception == FAILURE) return FAILURE;
305         else tekChainThrow(tek_exception);
306     }
307
308     // return the data stored at that node
309     *data = node_ptr->data;
310     return SUCCESS;
311 }
312 /**
313 * @brief Set the value of the hashtable at a certain key.
314 * @param hashtable A pointer to the hashtable.
315 * @param key The key to set the data at.
316 * @param data The actual data to set.
317 * @throws MEMORY_EXCEPTION if malloc() fails.
318 */
319 exception hashtableSet(HashTable* hashtable, const char* key,
320 void* data) {
321     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot set
null ptr..");
322
323     // check if hashtable needs to be rehashed before we add
the item
324     if (hashtableTooFull(hashtable)) {
325         tekChainThrow(hashtableRehash(hashtable, hashtable-
>length * 2));
326     }
327
328     // create a hash index
329     unsigned int hash;
330     tekChainThrow(hashtableHash(hashtable, key, &hash));
331
332     // get the node where the data is stored
333     HashNode* node_ptr;

```

```

334     // if there is no node, then create one
335     if (hashtableGetNode(hashtable, key, &node_ptr)) {
336         tekChainThrow(hashtableCreateNode(hashtable, key, hash,
337                                         &node_ptr));
337         hashtable->num_items += 1;
338     }
339
340     // set the data and update num of items
341     node_ptr->data = data;
342     return SUCCESS;
343 }
344
345 /**
346 * @brief Remove the data stored at a certain key in the
347 * hashtable.
348 * @note This does not free the pointer stored at this key,
349 * data stored here should be freed first.
350 * @param hashtable A pointer to the hashtable.
351 * @param key The key to remove.
352 * @throws HASHTABLE_EXCEPTION if the hashtable is empty.
353 * @throws FAILURE if the key does not exist.
354 */
355 exception hashtableRemove(HashTable* hashtable, const char*
key) {
356     if (!hashtable) tekThrow(NULL_PTR_EXCEPTION, "Cannot remove
from null ptr.");
357
358     // get the hash index
359     unsigned int hash;
360     tekChainThrow(hashtableHash(hashtable, key, &hash));
361
362     // iterate and keep track of current and previous nodes
363     HashNode* prev_ptr = 0;
364     HashNode* node_ptr = hashtable->internal[hash];
365     int found = FAILURE;
366     while (node_ptr) {
367         // if we found the key to remove
368         if (!strcmp(key, node_ptr->key)) {
369             found = SUCCESS;
370             // if there is a previous pointer, then push the
next pointer down the line
371             if (prev_ptr) {

```

```

371             prev_ptr->next = node_ptr->next;
372             // otherwise, there will be a new root node
373         } else {
374             hashtable->internal[hash] = node_ptr->next;
375         }
376
377         // free the node and the copy of the key
378         if (node_ptr->key) free(node_ptr->key);
379         free(node_ptr);
380
381         // update number of items
382         hashtable->num_items -= 1;
383         break;
384     }
385
386     // update pointers
387     prev_ptr = node_ptr;
388     node_ptr = node_ptr->next;
389 }
390 return found;
391 }
392
393 /**
394 * @brief Check whether an item exists in a hashtable.
395 * @param hashtable A pointer to the hashtable.
396 * @param key A key to an item to check
397 * @return 1 if the item exists, 0 if it does not or an error
occurred while looking for it
398 */
399 flag hashtableHasKey(HashTable* hashtable, const char* key) {
400     // attempt to get node, if fails then there is no node.
401     // kinda large overhead from writing exception buffer and
whatnot
402     HashNode* hash_node = 0;
403     const exception hashtable_result =
hashtableGetNode(hashtable, key, &hash_node);
404     if (hashtable_result == SUCCESS) return 1;
405     return 0;
406 }
407
408 /**
409 * @brief Print the data about a hashtable, which will print
something like

```

```

410 * @code
411 * HashNode* internal = 0x123456789
412 * unsigned int length = 100
413 * @endcode
414 * @param hashtable A pointer to a hashtable
415 */
416 void hashtablePrint(const HashTable* hashtable) {
417     // print
418     printf("HashNode* internal = %p\n", hashtable->internal);
419     printf("unsigned int length = %u\n", hashtable->length);
420 }
421
422 /**
423 * @brief Print a hashtable in a nice format, something like
424 * @code
425 * {
426 *     "key_1": 0x12987123
427 *     "key_2": 0x12312344
428 *     "key_3": 0x11245453
429 * }
430 * @endcode
431 * @param hashtable A pointer to a hashtable.
432 */
433 void hashtablePrintItems(const HashTable* hashtable) {
434     printf("%p\n", hashtable);
435     if (!hashtable) { // if no pointer, ofc cannot print
anything
436         printf("hashtable = NULL !");
437         return;
438     }
439     if (!hashtable->internal) { // no data
440         printf("hashtable->internal = NULL !");
441         return;
442     }
443     printf("{\n");
444     for (unsigned int i = 0; i < hashtable->length; i++) { // loop through and print each item in the internal array
445         const HashNode* node_ptr = hashtable->internal[i];
446         while (node_ptr) {
447             printf("    \"%s\" = %ld (hash=%d)\n", node_ptr->key, (long)node_ptr->data, i);
448             node_ptr = node_ptr->next;
449         }

```

```

450      }
451      printf("}\n");
452  }
453
454 /**
455  * @brief Print out the internal array of the hashtable, which
456  * will be a mix of 0s and pointers to linked lists.
457  * @note This is completely useless! (Except for when I was
458  * debugging this)
459  * @param hashtable A pointer to a hashtable.
460  */
461 void hashtablePrintInternal(const HashTable* hashtable) {
462     printf("{\n");
463     // code predating the "typedef uint" in tekgl.h!!!
464     // estimated time of writing: january 2025
465     for (unsigned int i = 0; i < hashtable->length; i++) {
466         printf("    internal[%u] = %p\n", i, hashtable-
467             >internal[i]);
468     }
469     printf("}\n");
470 }
```

core/hashtable.h

```

1 #pragma once
2
3 #include "exception.h"
4 #include "../tekgl.h"
5
6 typedef struct HashNode {
7     char* key;
8     void* data;
9     struct HashNode* next;
10 } HashNode;
11
12 typedef struct HashTable {
13     HashNode** internal;
14     unsigned int length;
15     unsigned int num_items;
16 } HashTable;
17
18 void hashtableDelete(const HashTable* hashtable);
19 exception hashtableCreate(HashTable* hashtable, unsigned int
length);
```

```

20 exception hashtableGetKeys(const HashTable* hashtable, char*** keys);
21 exception hashtableGetValues(const HashTable* hashtable, void*** values);
22 exception hashtableGet(HashTable* hashtable, const char* key,
void** data);
23 exception hashtableSet(HashTable* hashtable, const char* key,
void* data);
24 exception hashtableRemove(HashTable* hashtable, const char*
key);
25 flag hashtableHasKey(HashTable* hashtable, const char* key);
26 void hashtablePrint(const HashTable* hashtable);
27 void hashtablePrintItems(const HashTable* hashtable);
28 void hashtablePrintInternal(const HashTable* hashtable);

```

core/list.c

```

1 #include "list.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 /**
7  * Create a list using an existing but empty List struct.
8  * @note Not strictly needed, but recommended incase struct was
allocated using malloc() and contains bogus data.
9  * @param list A pointer to the list to initialise.
10 */
11 void listCreate(List* list) {
12     // just make sure that the list is empty
13     list->data = 0;
14     list->length = 0;
15 }
16
17 /**
18  * Delete a list by freeing all allocated nodes.
19  * @note Does not free any of the data added to the list, only
the list structure itself. Anything stored in the list should be
manually freed, or with \ref listFreeAllData'
20  * @param list The list to delete.
21 */
22 void listDelete(List* list) {
23     // iterate over each item
24     ListItem* item = list->data;

```

```

25
26     // if item is 0, it is the last item of the list
27     while (item) {
28         // we need to save the pointer to next, so it isn't
lost when item is freed
29         ListItem* next = item->next;
30         free(item);
31         item = next;
32     }
33     // clear the list so it can't be accidentally used
34     list->data = 0;
35     list->length = 0;
36 }
37
38 /**
39 * Call 'free(...)' on all data in the list.
40 * @param list The list to operate on.
41 */
42 void listFreeAllData(const List* list) {
43     ListItem* item = list->data;
44     while (item) {
45         // free each data ptr, and set it to 0 to avoid
accidental usage
46         free(item->data);
47         item->data = 0;
48         item = item->next;
49     }
50 }
51
52 /**
53 * Add an item to the list.
54 * @note Will only copy the pointer into the list, but not
actually the data. So don't try and add stack variables to it and
then pass on to another function, etc.
55 * @param list The list to add the item to.
56 * @param data The data/ptr to add to the list.
57 * @throws MEMORY_EXCEPTION if a new node could not be
allocated for the list.
58 */
59 exception listAddItem(List* list, void* data) {
60     // iterator variables
61     // also get a pointer to next pointer so we can overwrite
it

```

```

62     ListItem* item = list->data;
63     ListItem** item_ptr = &list->data;
64
65     // item can be 0, this signals an empty list that we cannot
66     // iterate over
67     if (item) {
68         // when next is 0, this is the last item
69         while (item->next) {
70             item = item->next;
71         }
72         // update overwriting pointer
73         item_ptr = &item->next;
74     }
75
76     // overwrite pointer with new item
77     *item_ptr = (ListItem*)malloc(sizeof(ListItem));
78     if (!*item_ptr) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for list.");
79
80     // fill new item with data
81     (*item_ptr)->data = data;
82
83     // next is 0, signalling no next item (this is now the last
84     // item)
85     (*item_ptr)->next = 0;
86
87     // increment size of list
88     list->length++;
89
90 /**
91  * Update the data stored at an index of a list.
92  * @note Data is not copied, only stored as a pointer.
93  * @param list The list to set an item of.
94  * @param index The index to set.
95  * @param data The data to be set at that index.
96  * @throws LIST_EXCEPTION if index out of range.
97  */
98 exception listItemSet(List* list, const uint index, void* data)
{
99     // obviously we cant insert an item outside of the list

```

```

100     if (index >= list->length) tekThrow(LIST_EXCEPTION, "List
index out of range.");
101
102     // create some iterator variables
103     ListItem* item = list->data;
104
105     // keep a pointer to the pointer to next so we can
overwrite it
106     ListItem** item_ptr = &list->data;
107
108     // if index is 0, then we are inserting at the start
109     // this means changing the root item of the list
110     // no need to run this loop
111     if (index) {
112         // counter variable
113         uint count = 0;
114         while (item->next) {
115             // if we have reached the specified index, then
stop looping
116             if (++count >= index) break;
117             item = item->next;
118         }
119
120         // update pointers
121         item_ptr = &item->next;
122     }
123
124     // fill item with new data
125     (*item_ptr)->data = data;
126
127     return SUCCESS;
128 }
129
130 /**
131 * Insert an item at a specific index into the list.
132 * @note Will only copy the pointer into the list, but not
actually the data. So don't try and add stack variables to it and
then pass on to another function, etc.
133 * @param list The list to insert an item into.
134 * @param index The index to insert the item at.
135 * @param data The data to be inserted into the list.
136 * @throws LIST_EXCEPTION if index out of range
137 * @throws MEMORY_EXCEPTION if malloc() fails.

```

```

138  */
139 exception listInsertItem(List* list, const uint index, void*
data) {
140     // obviously we cant insert an item outside of the list
141     if (index > list->length) tekThrow(LIST_EXCEPTION, "List
index out of range.");
142
143     // create some iterator variables
144     ListItem* item = list->data;
145
146     // at the start, the next item is the first item
147     ListItem* next = list->data;
148
149     // keep a pointer to the pointer to next so we can
overwrite it
150     ListItem** item_ptr = &list->data;
151
152     // if index is 0, then we are inserting at the start
153     // this means changing the root item of the list
154     // no need to run this loop, as it will change 'next'
incorrectly to the 1st item rather than 0th item
155     if (index) {
156         // counter variable
157         uint count = 0;
158         while (item->next) {
159             // if we have reached the specified index, then
stop looping
160             if (++count >= index) break;
161             item = item->next;
162         }
163
164         // update pointers
165         item_ptr = &item->next;
166         next = item->next;
167     }
168
169     // write new list item
170     *item_ptr = (ListItem*)malloc(sizeof(ListItem));
171     if (!*item_ptr) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for list.");
172
173     // fill item with data, using 'next' we stored
174     (*item_ptr)->data = data;

```

```

175     (*item_ptr)->next = next;
176
177     // increment length
178     list->length++;
179     return SUCCESS;
180 }
181
182 /**
183 * Get an item from the list.
184 * @param list The list to get an item from.
185 * @param index The index of the item to retrieve.
186 * @param data A pointer to where the data pointer will be
187 stored.
188 * @throws LIST_EXCEPTION if index is out of range.
189 */
190 exception listItem(const List* list, const uint index,
191 void** data) {
192     // make sure we are getting something that exists in the
193     list
194     if (index >= list->length) tekThrow(LIST_EXCEPTION, "List
195 index out of range.");
196
197     // iterator variables
198     const ListItem* item = list->data;
199     uint count = 0;
200     while (item) {
201         // if we have counted up to the index, stop
202         if (count++ >= index) break;
203         item = item->next;
204     }
205 }
206
207 /**
208 * Pop an item from the end of the list.
209 * @param list The list to pop an item from.
210 * @param data A pointer to where the popped data will be
211 stored.
212 * @throws LIST_EXCEPTION if the list is empty.
213 */

```

```

213 exception listPopItem(List* list, void** data) {
214     // track current and previous item in loop
215     ListItem* prev = 0;
216     ListItem* item = list->data;
217
218     // if item is 0, then there's nothing to pop
219     if (!item) tekThrow(LIST_EXCEPTION, "List is empty.");
220     while (item) {
221         // loop until there is no next item
222         if (!item->next) break;
223
224         // keep track of previous item before moving on
225         prev = item;
226         item = item->next;
227     }
228
229     // return the original data before we free it
230     if (data) *data = item->data;
231     free(item);
232
233     // if there is not a previous item, then we have just
234     // emptied the list completely
235     // hence the two possibilities - editing one item or the
236     // actual list
237     if (prev) {
238         prev->next = 0;
239     } else {
240         list->data = 0;
241
242         // decrement length
243         list->length--;
244         return SUCCESS;
245     }
246 /**
247 * Remove an item from the list.
248 * @param list The list to remove an item from.
249 * @param index The index of the item that should be removed.
250 * @param data A pointer to where the data pointer should be
251 * stored.
252 * @throws LIST_EXCEPTION if the index is out of range.
253 */

```

```

253 exception listRemoveItem(List* list, const uint index, void** data) {
254     // track previous and current item
255     ListItem* prev = 0;
256     ListItem* item = list->data;
257
258     // make sure that the list is not empty, and we are
259     // removing an existing item
260     if (!item) tekThrow(LIST_EXCEPTION, "List is empty.");
261     if (index >= list->length) tekThrow(LIST_EXCEPTION, "List
262 index out of range.");
263
264     // iterate over list
265     uint count = 0;
266     while (item) {
267         // if we have hit the index to remove, stop looping
268         if (count++ >= index) break;
269
270         // update iterator variables
271         prev = item;
272         item = item->next;
273     }
274
275     // return data before we free it
276     if (data) *data = item->data;
277
278     // pass the next pointer to the previous item, so the list
279     // remains contiguous
280     // could be either the previous item, or the root of the
281     // list
282     if (prev) {
283         prev->next = item->next;
284     } else {
285         list->data = item->next;
286     }
287
288     // free item, as it has been removed
289     free(item);
290     list->length--;
291     return SUCCESS;
292 }
```

```

291
292  /**
293   * Move an item at one index so that it is now stored at a
294   * different index.
295   * @param list The list for which to move the items in.
296   * @param old_index The index of the item to be moved.
297   * @param new_index The index which the item should be moved
298   * into.
299   * @throws LIST_EXCEPTION if either index is out of range.
300   */
301 exception listMoveItem(List* list, const uint old_index, const
302   uint new_index) {
303     // make sure that we actually need to move the item
304     if (old_index == new_index) return SUCCESS;
305
306     // scan the list to find the old item
307     ListItem* item = list->data, * prev = 0;
308     uint index = 0;
309     while (item) {
310       if (index == old_index) break;
311       index++;
312       prev = item;
313       item = item->next;
314     }
315
316     // plug the gap between the two items either side of the
317     // moved item.
318     if (old_index == 0) {
319       list->data = item->next;
320     } else {
321       prev->next = item->next;
322     }
323     // keep track of what the old item was.
324     ListItem* moved_item = item;
325
326     // if inserting to be the new first item, then need to
327     // directly set as the root item.
328     if (new_index == 0) {

```

```

328         moved_item->next = list->data;
329         list->data = moved_item;
330         return SUCCESS;
331     }
332
333     // if the old index is greater than the new index, the
334     // search should restart from the beginning
335     // if the new index is after the old one, we need not start
336     // from the beginning and can just continue from where we got to.
337     if (old_index > new_index || old_index == 0) {
338         prev = 0;
339         item = list->data;
340         index = 0;
341     }
342
343     // iterate to find the new index.
344     while (item) {
345         prev = item;
346         item = item->next;
347         if (index == new_index) break;
348         index++;
349     }
350
351     // insert the item into the new location.
352     prev->next = moved_item;
353     moved_item->next = item;
354
355     return SUCCESS;
356 }
357 /**
358 * Print out the data pointer of each item in the list in
359 * hexadeciml format.
360 * @note Not very useful unless debugging. For more useful
361 * output, your own function is needed as the list has no idea what
362 * datatype you are storing.
363 * @param list The list to print out.
364 */
365 void listPrint(const List* list) {
366     // iterate over list
367     const ListItem* item = list->data;
368     uint count = 0;
369     printf("[");

```

```

366     while (item) {
367         // if not the first item, we need commas to separate
368         items
369         if (count) printf(", ");
370         // print the item
371         printf("%p", item->data);
372         item = item->next;
373         count++;
374     }
375     printf("]\n");
376 }
```

core/list.h

```

1 #pragma once
2
3 #include "exception.h"
4 #include "../tekgl.h"
5
6 #define foreach(item, list, run) \
7     item = list->data; \
8     while (item) { \
9         run; \
10        item = item->next; \
11    } \
12
13 typedef struct ListItem {
14     void* data;
15     struct ListItem* next;
16 } ListItem;
17
18 typedef struct List {
19     ListItem* data;
20     uint length;
21 } List;
22
23 void listCreate(List* list);
24 void listDelete(List* list);
25 void listFreeAllData(const List* list);
26 exception listAddItem(List* list, void* data);
27 exception listSetItem(List* list, uint index, void* data);
28 exception listInsertItem(List* list, uint index, void* data);
```

```

29 exception listGetItem(const List* list, uint index, void** data);
30 exception listPopItem(List* list, void** data);
31 exception listRemoveItem(List* list, uint index, void** data);
32 exception listMoveItem(List* list, uint old_index, uint new_index);
33 void listPrint(const List* list);

```

core/priorityqueue.c

```

1 #include "priorityqueue.h"
2
3 #include <stdlib.h>
4
5 /**
6  * @brief Initialise a priority queue struct.
7  * @note This function requires the priority queue struct to be
allocated already.
8  * @param queue A pointer to an empty queue.
9  */
10 void priorityQueueCreate(PriorityQueue* queue) {
11     // default values
12     queue->queue = 0;
13     queue->length = 0;
14 }
15
16 /**
17  * @brief Free all memory that a priority queue has allocated.
18  * @param queue The priority queue to free.
19  */
20 void priorityQueueDelete(PriorityQueue* queue) {
21     PriorityQueueItem* item = queue->queue;
22     while (item) {
23         // essentially, go to each item starting from the
front, store the item before it, then free itself.
24         PriorityQueueItem* prev = item->next;
25         free(item);
26         item = prev;
27     }
28     // zero the struct to avoid misuse of freed pointers.
29     queue->queue = 0;
30     queue->length = 0;
31 }
32

```

```

33 /**
34  * Enqueue a piece of data into the priority queue. Data with
35  * the lowest priority value will be dequeued first.
36  * @param queue The priority queue to enqueue into.
37  * @param priority The priority value given to the data.
38  * @param data A pointer to the data to be stored.
39  * @throws MEMORY_EXCEPTION if a new node for the queue could
40  * not be created.
41  */
42 exception priorityQueueEnqueue(PriorityQueue* queue, const
43 double priority, void* data) {
44     PriorityQueueItem* new_item =
45     (PriorityQueueItem*)malloc(sizeof(PriorityQueueItem));
46     if (!new_item)
47         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
48 for priority queue item.");
49     new_item->priority = priority;
50     new_item->data = data;
51     new_item->next = 0;
52
53     queue->length++;
54
55     // if the queue is uninitialised, then this is the first
56     // item, so must be the front of the list.
57     if (!queue->queue) {
58         queue->queue = new_item;
59         return SUCCESS;
60     }
61
62     // if the priority is smaller than the first item, this now
63     // becomes the first item.
64     if (priority < queue->queue->priority) {
65         new_item->next = queue->queue;
66         queue->queue = new_item;
67         return SUCCESS;
68     }
69
70     // otherwise, iterate over the list to find the first item
71     // that has a bigger priority than the new item.
72     // or if no item is found, give the last item in the queue.
73     PriorityQueueItem* item = queue->queue;
74     while (item->next) {
75         if (priority < item->next->priority)

```

```

68         break;
69         item = item->next;
70     }
71
72     // insert item into the queue at this point.
73     new_item->next = item->next;
74     item->next = new_item;
75     return SUCCESS;
76 }
77
78 /**
79  * Dequeue a piece of data from the front of the priority
queue.
80  * @param queue The queue to dequeue from.
81  * @param data A pointer to where the data can be stored. Will
be set to 0 if the queue is empty.
82  * @return 0 if the queue is empty, 1 otherwise.
83  */
84 flag priorityQueueDequeue(PriorityQueue* queue, void** data) {
85     // if no front pointer, queue is empty.
86     if (!queue->queue) {
87         *data = 0;
88         return 0;
89     }
90
91     // shift everything in queue forwards and free the old
front.
92     PriorityQueueItem* front = queue->queue;
93     *data = front->data;
94     queue->queue = front->next;
95     free(front);
96
97     queue->length--;
98
99     return 1;
100 }
101
102 /**
103  * Peek an item at the front of the priority queue.
104  * @param queue The queue to peek into.
105  * @param data The data that was found at the front.
106  * @return 0 if the queue is empty, 1 otherwise.
107  */

```

```

108 flag priorityQueuePeek(const PriorityQueue* queue, void** data)
{
109     if (!queue->queue) { // if queue empty, cannot peek any
data
110         *data = 0;
111         return 0;
112     }
113     *data = queue->queue->data; // get data at the front of the
queue.
114     return 1;
115 }
116
117 /**
118 * @brief Check whether a priority queue is empty.
119 * @param queue The priority queue to check.
120 * @return 0 if the queue is empty, 1 if it is not empty.
121 */
122 flag priorityQueueIsEmpty(const PriorityQueue* queue) {
123     // if there is no front of the queue, then the queue must
be empty.
124     return queue->queue ? 0 : 1;
125 }
```

core/priorityqueue.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "exception.h"
5
6 typedef struct PriorityQueueItem {
7     double priority;
8     void* data;
9     struct PriorityQueueItem* next;
10 } PriorityQueueItem;
11
12 typedef struct PriorityQueue {
13     PriorityQueueItem* queue;
14     uint length;
15 } PriorityQueue;
16
17 void priorityQueueCreate(PriorityQueue* queue);
18 void priorityQueueDelete(PriorityQueue* queue);
```

```

19 exception priorityQueueEnqueue(PriorityQueue* queue, double
priority, void* data);
20 flag priorityQueueDequeue(PriorityQueue* queue, void** data);
21 flag priorityQueuePeek(const PriorityQueue* queue, void** data);
22 flag priorityQueueIsEmpty(const PriorityQueue* queue);

```

core/queue.c

```

1 #include "queue.h"
2
3 #include <stdlib.h>
4
5 /**
6  * @brief Initialise a queue struct.
7  * @note This function requires the queue struct to be
allocated already.
8  * @param queue A pointer to an empty queue.
9  */
10 void queueCreate(Queue* queue) {
11     // null out everything incase bogus data was in the struct
12     queue->rear = 0;
13     queue->front = 0;
14     queue->length = 0;
15 }
16
17 /**
18  * @brief Free all memory that a queue has allocated.
19  * @param queue The queue to free.
20  */
21 void queueDelete(Queue* queue) {
22     QueueItem* item = queue->front;
23     while (item) {
24         // essentially, go to each item starting from the
front, store the item before it, then free itself.
25         QueueItem* prev = item->next;
26         free(item);
27         item = prev;
28     }
29     // zero the struct to avoid misuse of freed pointers.
30     queue->rear = 0;
31     queue->front = 0;
32     queue->length = 0;
33 }
34

```

```

35 /**
36  * @brief Add a piece of data to the queue.
37  * @param queue The queue to enqueue into.
38  * @param data The data to enqueue.
39  * @throws MEMORY_EXCEPTION if memory could not be allocated
for the queue.
40 */
41 exception queueEnqueue(Queue* queue, void* data) {
42     // we are going to overwrite the last item of the queue.
43     // so make sure to store what was there before we overwrite
it.
44     QueueItem* item = queue->rear;
45     queue->rear = (QueueItem*)malloc(sizeof(QueueItem));
46     if (!queue->rear)
47         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for queue.");
48     queue->rear->data = data;
49
50     // as this is the last item, there is nothing before it in
the queue.
51     queue->rear->next = 0;
52
53     // depending on whether there was a last item before we
added this one, we need to act differently.
54     if (item) {
55         // if we have changed the last item, then the old last
item has now got a previous item to point to, which is this new one.
56         item->next = queue->rear;
57     } else {
58         //if there wasn't a last item before, this new item is
also the first item (as it is the only item right now)
59         queue->front = queue->rear;
60     }
61     queue->length++;
62     return SUCCESS;
63 }
64
65 /**
66  * @brief Retrieve the data stored at the front of the queue,
and remove it from the queue.
67  * @param queue The queue to dequeue from.
68  * @param data A pointer to where the retrieved data should be
stored.

```

```

69  * @throws QUEUE_EXCEPTION if the queue is empty.
70  */
71 exception queueDequeue(Queue* queue, void** data) {
72     // if there is nothing at the front, then we can't dequeue
73     if (!queue->front)
74         tekThrow(QUEUE_EXCEPTION, "Cannot dequeue from an empty
queue.");
75     *data = queue->front->data;
76
77     // set the front to the item before the old front.
78     QueueItem* prev = queue->front->next;
79     free(queue->front);
80     queue->front = prev;
81
82     // if there is no longer a front of the queue, then that
means we cleared the last item.
83     if (!queue->front) queue->rear = 0;
84     queue->length--;
85     return SUCCESS;
86 }
87
88 /**
89  * @brief Retrieve the data stored at the front of the queue.
90  * @param queue The queue to be peeked.
91  * @param data A pointer to where the retrieved data should be
stored.
92  * @throws QUEUE_EXCEPTION if the queue is empty.
93  */
94 exception queuePeek(const Queue* queue, void** data) {
95     if (!queue->front)
96         tekThrow(QUEUE_EXCEPTION, "Cannot peek an empty
queue.");
97     *data = queue->front->data; // root node = front of queue
98     return SUCCESS;
99 }
100
101 /**
102  * @brief Check whether a queue is empty.
103  * @param queue The queue to check.
104  * @return 0 if the queue is empty, 1 if it is not empty.
105  */
106 flag queueIsEmpty(const Queue* queue) {

```

```

107      // if there is no front of the queue, then the queue must
be empty.
108      return queue->front ? 0 : 1;
109  }

```

core/queue.h

```

1  #pragma once
2
3  #include "../tekgl.h"
4  #include "exception.h"
5
6  typedef struct QueueItem {
7      void* data;
8      struct QueueItem* next;
9  } QueueItem;
10
11 typedef struct Queue {
12     QueueItem* front;
13     QueueItem* rear;
14     uint length;
15 } Queue;
16
17 void queueCreate(Queue* queue);
18 void queueDelete(Queue* queue);
19 exception queueEnqueue(Queue* queue, void* data);
20 exception queueDequeue(Queue* queue, void** data);
21 exception queuePeek(const Queue* queue, void** data);
22 flag queueIsEmpty(const Queue* queue);

```

core/stack.c

```

1  #include "stack.h"
2
3  #include <stdlib.h>
4
5  /**
6   * Create a stack from an existing struct.
7   * @note Not strictly necessary, essentially just zeroes out the
struct.
8   * @param stack The stack to initialise.
9   */
10 void stackCreate(Stack* stack) {
11     // just make sure that the stack is empty
12     stack->data = 0;

```

```

13     stack->length = 0;
14 }
15
16 /**
17 * Delete a stack, by freeing all nodes of the stack.
18 * @note Doesn't free any of the stored data, just the structure
19 * itself. Freeing data should be done manually.
20 */
21 void stackDelete(Stack* stack) {
22     // iterate over each item
23     StackItem* item = stack->data;
24
25     // if item is 0, it is the last item of the stack
26     while (item) {
27         // we need to save the pointer to next, so it isn't lost
when item is freed
28         StackItem* next = item->next;
29         free(item);
30         item = next;
31     }
32     // clear the stack so it can't be accidentally used
33     stack->data = 0;
34     stack->length = 0;
35 }
36
37 /**
38 * Push an item onto the stack.
39 * @note This will not copy the data that the pointer points to,
it will only store the pointer. So beware of the pointer being freed
while stack still exists.
40 * @param stack The stack to push an item onto.
41 * @param data Data / a pointer to be added.
42 * @throws MEMORY_EXCEPTION if a new node could not be
allocated.
43 */
44 exception stackPush(Stack* stack, void* data) {
45     // store the original pointer
46     StackItem* next = stack->data;
47
48     // keep a pointer to the start so we can overwrite it
49     StackItem** item_ptr = &stack->data;
50

```

```

51     // write new item
52     *item_ptr = (StackItem*)malloc(sizeof(StackItem));
53     if (!*item_ptr) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for stack.");
54
55     // fill item with data, using 'next' we stored
56     (*item_ptr)->data = data;
57     (*item_ptr)->next = next;
58
59     // increment length
60     stack->length++;
61     return SUCCESS;
62 }
63
64 /**
65  * Pop an item from the stack.
66  * @param stack The stack to pop from.
67  * @param data A pointer to where the data will be stored.
68  * @throws STACK_EXCEPTION if the stack is empty.
69 */
70 exception stackPop(Stack* stack, void** data) {
71     if (!stack->data) tekThrow(STACK_EXCEPTION, "Stack is
empty");
72
73     // return root node and set root node to child of root node.
74     StackItem* item = stack->data;
75     *data = item->data;
76     stack->data = item->next;
77     free(item);
78
79     // decrement length
80     stack->length--;
81     return SUCCESS;
82 }
83
84 /**
85  * Peek the stack / return item on the top of the stack without
removing it.
86  * @param stack The stack to peek.
87  * @param data A pointer to where the data should be stored.
88  * @throws STACK_EXCEPTION if the stack is empty.
89 */
90 exception stackPeek(const Stack* stack, void** data) {

```

```

91     if (!stack->data) tekThrow(STACK_EXCEPTION, "Stack is
empty");
92     *data = stack->data->data; // top of stack = root node
93
94     return SUCCESS;
95 }

```

core/stack.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "exception.h"
5
6 typedef struct StackItem {
7     void* data;
8     struct StackItem* next;
9 } StackItem;
10
11 typedef struct Stack {
12     StackItem* data;
13     uint length;
14 } Stack;
15
16 void stackCreate(Stack* stack);
17 void stackDelete(Stack* stack);
18 exception stackPush(Stack* stack, void* data);
19 exception stackPop(Stack* stack, void** data);
20 exception stackPeek(const Stack* stack, void** data);

```

core/testsuite.c

```
1 #include "testsuite.h"
```

core/testsuite.h

```

1 #include "../tekgl.h"
2 #include "exception.h"
3
4 #define MAX_TESTS 16
5
6 typedef exception(*TekTestFunc)(void* test_context);
7
8 /**
9  * Create a creation function for a suite. Called every test.
10 * @param test_name Name of the suite

```

```

11  */
12 #define tekTestCreate(test_name) exception
__tekCreate_##test_name
13 /**
14  * Create a cleanup function for a test suite.
15  * @param test_name Name of the suite
16  */
17 #define tekTestDelete(test_name) exception
__tekDelete_##test_name
18 /**
19  * Create a new function in a test suite.
20  * @param test_name Name of suite
21  * @param test_part Name of test
22  */
23 #define tekTestFunc(test_name, test_part) exception
__tekTest_##test_name##_##test_part
24
25 /**
26  * Run a suite of tests
27  * @param test_name The name of the suite
28  * @param test_part The function to run in that suite
29  * @param test_context The shared memory struct.
30  */
31 #define tekRunSuite(test_name, test_part, test_context) \
32 printf("Testing \"%s %s\":\n", #test_name, #test_part); \
33 tekChainThrow(__tekCreate_##test_name(test_context)); \
34
35 tekChainThrow(__tekTest_##test_name##_##test_part(test_context)); \
36 tekChainThrow(__tekDelete_##test_name(test_context)); \
37 /**
38  * Assert if two things are equal, and print if they are or are
not.
39  * @param expected
40  * @param actual
41  * @return
42  */
43 #define tekAssert(expected, actual) \
44 if ((expected) == (actual)) { \
45     printf("    Test Passed: %s == %s\n", #expected, #actual); \
46 } else { \
47     tekThrow(ASSERT_EXCEPTION, "Test Failed: " #expected " != " \
#actual "\n"); \

```

```

48 }
49
50 /**
51  * Assert if two things are equivalent, but only alert if they
52  * are not equal.
53  * @param expected
54  * @param actual
55  */
56 #define tekSilentAssert(expected, actual) \
57 if ((expected) != (actual)) { \
58     tekThrow(ASSERT_EXCEPTION, "Test Failed: " #expected " != " \
#actual "\n"); \
59 }

```

core/threadqueue.c

```

1  #include "threadqueue.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6 /**
7  * Initialise a thread queue. Based on the principle of a lock-
8  * free circular queue.
9  * @note Single-consumer single-producer
10 * @param thread_queue A pointer to an existing but empty
ThreadQueue struct.
11 * @param capacity The capacity of the thread queue.
12 * @throws MEMORY_EXCEPTION if malloc() fails.
13 */
14 exception threadQueueCreate(ThreadQueue* thread_queue, const
15 uint capacity) {
16     // allocate memory for queue
17     thread_queue->buffer = (void**)malloc(capacity *
18     sizeof(void*));
19     if (!thread_queue->buffer)
20         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
21 buffer for thread queue.");
22     thread_queue->buffer_size = capacity;
23     // atomic integer for front and rear
24     atomic_init(&thread_queue->front, 0);
25     atomic_init(&thread_queue->rear, 0);
26     return SUCCESS;

```

```

24 }
25 /**
26 * Delete a thread queue and free the buffer allocated.
27 * @note Only frees the thread queue's used memory, not that of
28 stored pointers.
29 * @param thread_queue The thread queue to delete.
30 */
31 void threadQueueDelete(ThreadQueue* thread_queue) {
32     // free allocated memory
33     free(thread_queue->buffer);
34
35     // prevent misuse by setting things to null
36     thread_queue->buffer = 0;
37     thread_queue->buffer_size = 0;
38 }
39
40 /**
41 * Enqueue a new item to the thread queue, storing a pointer.
42 * @note Only to be used by a single producer thread.
43 * @param thread_queue The thread queue to enqueue to.
44 * @param data A pointer to store in the queue.
45 * @returns 0 if the queue is already full, 1 if the operation
was successful.
46 */
47 flag threadQueueEnqueue(ThreadQueue* thread_queue, void* data)
{
    // can use relaxed memory order here, thread using this
"owns" the rear pointer, other thread should write after we are
done.
    const uint rear = atomic_load_explicit(&thread_queue->rear,
memory_order_relaxed);
    const uint next_rear = (rear + 1) % thread_queue-
>buffer_size;
    ...
    // need to use acquire here, should get front index after
the other thread has finished with it.
    // we only need this to make sure the queue isn't full.
    const uint front = atomic_load_explicit(&thread_queue-
>front, memory_order_acquire);
    ...
    // if front == next_rear, this write would overwrite data
in the queue

```

```

57     if (front == next_rear) {
58         return 0;
59     }
60
61     thread_queue->buffer[rear] = data;
62     // release order means that subsequent access to rear must
happen after this, so other thread has to have the updated rear ptr
63     atomic_store_explicit(&thread_queue->rear, next_rear,
memory_order_release);
64     return 1;
65 }
66
67 /**
68 * Dequeue from a thread queue, returning the stored pointer.
69 * @note Only to be used by a single consumer thread.
70 * @param thread_queue The thread queue to dequeue from.
71 * @param data A pointer to a pointer that will receive the
dequeued data.
72 * @returns 0 if the queue is empty, 1 if the operation was
successful.
73 */
74 flag threadQueueDequeue(ThreadQueue* thread_queue, void** data)
{
    // dequeue side is the main controller of the front ptr, we
can be relaxed because other thread is working around what we do
with it.
    const uint front = atomic_load_explicit(&thread_queue-
>front, memory_order_relaxed);
    // however, we need to ensure that other thread is finished
with rear ptr before we can use it
    const uint rear = atomic_load_explicit(&thread_queue->rear,
memory_order_acquire);
    if (front == rear, the queue is empty
81     if (front == rear) {
82         return 0;
83     }
84
85     *data = thread_queue->buffer[front];
86     // make sure that when front ptr is read next, it happens
after it has been updated to the new size
87     atomic_store_explicit(&thread_queue->front, (front + 1) %
thread_queue->buffer_size, memory_order_release);

```

```

88     return 1;
89 }
90
91 /**
92  * Peek into a thread queue, returning the stored pointer.
93  * @note Only to be used by a single consumer thread.
94  * @param thread_queue The thread queue to peek.
95  * @param data A pointer to a pointer that will receive the
peeked data.
96  * @returns 0 if the queue is empty, 1 if the operation was
successful.
97 */
98 flag threadQueuePeek(ThreadQueue* thread_queue, void** data) {
99     // dequeue side is the main controller of the front ptr, we
can be relaxed because other thread is working around what we do
with it.
100    const uint front = atomic_load_explicit(&thread_queue-
>front, memory_order_relaxed);
101    // however, we need to ensure that other thread is finished
with rear ptr before we can use it
102    const uint rear = atomic_load_explicit(&thread_queue->rear,
memory_order_acquire);
103
104    // if front == rear, the queue is empty
105    if (front == rear) {
106        return 0;
107    }
108
109    *data = thread_queue->buffer[front];
110    return 1;
111 }
112
113 /**
114  * Determine whether a queue is empty.
115  * @note Only to be used by a single consumer thread.
116  * @param thread_queue The thread queue to check.
117  * @returns 1 if the queue is empty, 0 if not.
118 */
119 flag threadQueueIsEmpty(ThreadQueue* thread_queue) {
120     // dequeue side is the main controller of the front ptr, we
can be relaxed because other thread is working around what we do
with it.

```

```

121     const uint front = atomic_load_explicit(&thread_queue-
>front, memory_order_relaxed);
122     // however, we need to ensure that other thread is finished
with rear ptr before we can use it
123     const uint rear = atomic_load_explicit(&thread_queue->rear,
memory_order_acquire);
124
125     return front == rear;
126 }
```

core/threadqueue.h

```

1 #pragma once
2
3 #include <stdatomic.h>
4 #include <semaphore.h>
5
6 #include "../tekgl.h"
7 #include "exception.h"
8
9 typedef struct ThreadQueue {
10     void** buffer;
11     uint buffer_size;
12     atomic_uint front;
13     atomic_uint rear;
14 } ThreadQueue;
15
16 exception threadQueueCreate(ThreadQueue* thread_queue, uint
capacity);
17 void threadQueueDelete(ThreadQueue* thread_queue);
18 flag threadQueueEnqueue(ThreadQueue* thread_queue, void* data);
19 flag threadQueueDequeue(ThreadQueue* thread_queue, void** data);
20 flag threadQueuePeek(ThreadQueue* thread_queue, void** data);
21 flag threadQueueIsEmpty(ThreadQueue* thread_queue);
```

core/vector.c

```

1 #include "vector.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 /**
8 * Create a vector / resizing array
```

```

9   * @param start_capacity The number of items that the vector
can initially store (use 1 if unsure)
10  * @param element_size The size of each element, use
'sizeof(TheTypeIWannaUse)' here.
11  * @param vector A pointer to a Vector struct to be used.
12  * @throws VECTOR_EXCEPTION if the element size is 0
13  * @throws MEMORY_EXCEPTION if the initial list could not be
allocated.
14  */
15 exception vectorCreate(uint start_capacity, const uint
element_size, Vector* vector) {
16     // mostly just safety checks here
17     if (element_size == 0) tekThrow(VECTOR_EXCEPTION, "Vector
elements cannot have a size of 0.");
18     if (start_capacity == 0) start_capacity = 1; // if start
capacity is 0, it would infinitely try and double the size of the
vector cuz  $2 * 0 = 0$ . and would never grow. just seemed too harsh to
throw an error for that when it really has no effect.
19     void* internal = (void*)malloc(start_capacity *
element_size);
20     if (!internal) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate initial memory for vector.");
21     vector->internal = internal;
22     vector->internal_size = start_capacity;
23     vector->element_size = element_size;
24     vector->length = 0;
25     return SUCCESS;
26 }
27
28 /**
29 * Directly write an item to the vector.
30 * @note Doesn't perform any safety checks, only used
internally to avoid repeating code.
31 * @param vector The vector to write into.
32 * @param index The index of the element to write.
33 * @param item The item to write into the vector.
34 */
35 static void vectorWriteItem(const Vector* vector, const uint
index, const void* item) {
36     // calculate byte array index and copy bytes
37     void* dest = vector->internal + index * vector-
>element_size;
38     memcpy(dest, item, vector->element_size);

```

```

39 }
40 /**
41 * Increase the capacity of the vector by a factor of 2.
42 * @param vector The vector to increase the size of.
43 * @throws MEMORY_EXCEPTION if extra memory could not be
44 allocated.
45 */
46 static exception vectorDoubleCapacity(Vector* vector) {
47     // double the internal size
48     const uint new_size = vector->internal_size * 2;
49     void* internal = vector->internal;
50     void* temp = (void*)realloc(internal, new_size * vector-
>element_size);
51
52     // attempt to realloc
53     if (!temp) tekThrowThen(MEMORY_EXCEPTION, "Failed to
allocate more memory to grow vector.", free(internal));
54     // if worked, then we can set the new data in the vector
55     // if we did it earlier, then it would crash trying to
delete the vector
56     vector->internal = temp;
57     vector->internal_size = new_size;
58     return SUCCESS;
59 }
60 /**
61 * Add an item to the vector.
62 * @note The item WILL be copied into the vector, you can
safely free anything added to vector afterwards and keep the data
here.
63 * @param vector The vector to add the item to.
64 * @param item The item that should be copied into the vector.
65 * @throws MEMORY_EXCEPTION if the vector required a resize and
this failed.
66 */
67 exception vectorAddItem(Vector* vector, const void* item) {
68     if (vector->length >= vector->internal_size)
69         tekChainThrow(vectorDoubleCapacity(vector)); // amazing
helper function
70     vectorWriteItem(vector, vector->length, item); // add item
at end

```

```

72     vector->length++; // increment length cuz its longer now by
1
73     return SUCCESS;
74 }
75
76 /**
77 * Set an item in the vector at a certain index.
78 * @note The item WILL be copied into the vector, you can
safely free anything added to vector afterwards and keep the data
here.
79 * @param vector The vector to operate on.
80 * @param index The index of the item that should be set.
81 * @param item The new data to store at this index.
82 * @throws VECTOR_EXCEPTION if the index is out of bounds.
83 */
84 exception vectorSetItem(const Vector* vector, const uint index,
const void* item) {
85     // just add a check to the write function
86     if (index >= vector->length) tekThrow(VECTOR_EXCEPTION,
"Attempted to set index out of bounds.");
87     vectorWriteItem(vector, index, item);
88     return SUCCESS;
89 }
90
91 /**
92 * Get an item from the vector.
93 * @note The pointer to the item shoud be a buffer large enough
to contain the retrieved item. Works best to use the struct you used
when setting the element size at initialisation.
94 * @param vector The vector to get an item from.
95 * @param index The index of the item to get from the vector.
96 * @param item A pointer to where the item should be stored.
97 * @throws VECTOR_EXCEPTION if the index is out of bounds.
98 */
99 exception vectorGetItem(const Vector* vector, const uint index,
void* item) {
100     if (index >= vector->length) tekThrow(VECTOR_EXCEPTION,
"Attempted to get index out of bounds.");
101
102     // copy data into buffer from user
103     const void* src = vector->internal + index * vector-
>element_size;
104     memcpy(item, src, vector->element_size);

```

```

105     return SUCCESS;
106 }
107
108 /**
109  * Get the pointer to an item in the list.
110 * @note Can be better than using \ref vectorGetItem as it
avoids making a copy of the data. But if a copy is needed, use the
other version.
111 * @param vector The vector to get the item pointer from.
112 * @param index The index of the item to get.
113 * @param item A pointer to where the item pointer should be
stored.
114 * @throws VECTOR_EXCEPTION if the index is out of bounds.
115 */
116 exception vectorGetItemPtr(const Vector* vector, const uint
index, void** item) {
117     if (index >= vector->length) tekThrow(VECTOR_EXCEPTION,
"Attempted to get index out of bounds.");
118
119     // basic pointers stuff (yeah)
120     *item = vector->internal + index * vector->element_size;
121     return SUCCESS;
122 }
123
124 /**
125  * Remove an item from the vector.
126 * @note This will shift all the other items in the vector down
to fill the gap that is left. If you want to avoid this, just use \ref
vectorSetItem with a zeroed struct.
127 * @param vector The vector to remove the item from.
128 * @param index The index of the item to be removed.
129 * @param item A pointer to where the removed item should be
stored. Needs to point to a block of memory big enough to store the
item. Set to NULL if item is not needed.
130 * @throws VECTOR_EXCEPTION if the index is out of bounds.
131 */
132 exception vectorRemoveItem(Vector* vector, const uint index,
void* item) {
133     if (index >= vector->length) tekThrow(VECTOR_EXCEPTION,
"Attempted to get index out of bounds.");
134
135     // get copy of the item before its gone (if wanted)

```

```

136     if (item) tekChainThrow(vectorGetItem(vector, index,
137     item));
138     // shift everything after the item being removed down by
139     one item
140     void* dest = vector->internal + index * vector-
141     >element_size;
142     const void* src = dest + vector->element_size;
143     const uint remainder_size = (vector->length - index - 1) *
144     vector->element_size;
145     memcpy(dest, src, remainder_size);
146     vector->length--;
147     return SUCCESS;
148 }
149 /**
150 * Pop an item from the vector, e.g. return last item and
151 remove it from the vector.
152 * @param vector The vector to pop from.
153 * @param item A pointer to where the item will be stored. Must
154 point to a buffer large enough to store the returned data.
155 * @note Internally uses \ref vectorGetItem, check for more
156 info.
157 * @returns 1 if successful, 0 if the vector is empty.
158 */
159 flag vectorPopItem(Vector* vector, void* item) {
160     if (vector->length == 0) {
161         memset(item, 0, vector->element_size);
162         return 0;
163     };
164     // allowed to not check this error, can only fail if index
165     out of range, and the by using length - 1, it must be in range.
166     // don't need to use vectorRemoveItem(...) because we are
167     taking out the last item, so nothing needs to be moved around in the
168     vector.
169     // new writes should overwrite the old data.
170     vectorGetItem(vector, vector->length - 1, item);
171     vector->length--;
172     return 1;
173 }
174

```

```

169 /**
170  * Insert an item into a vector at a certain index.
171  * @param vector The vector in which to insert the item.
172  * @param index The index at which to insert the new item.
173  * @param item The item which is to be inserted.
174  * @throws VECTOR_EXCEPTION if the index is out of bounds.
175  * @throws MEMORY_EXCEPTION if malloc() failed.
176 */
177 exception vectorInsertItem(Vector* vector, const uint index,
178 const void* item) {
179     if (index > vector->length) tekThrow(VECTOR_EXCEPTION,
180 "Attempted to set index out of bounds.");
181
182     // new item to be added to vector at the end, just use the
183     // tried and true add method
184     if (index == vector->length) {
185         tekChainThrow(vectorAddItem(vector, item));
186         return SUCCESS;
187     }
188
189     // if adding past the end, need to grow the size
190     if (vector->length >= vector->internal_size)
191         tekChainThrow(vectorDoubleCapacity(vector));
192
193     // mega bodge
194     // shift everything at index and after one item forward
195     void* src = vector->internal + index * vector-
196     >element_size;
197     void* dest = src + vector->element_size;
198     const uint size = (vector->length - index) * vector-
199     >element_size;
200     memcpy(dest, src, size);
201
202     // now add the new item
203     vectorWriteItem(vector, index, item);
204     vector->length++;
205     return SUCCESS;
206 }
207
208 /**
209  * Set all items in the vector to NULL and reset the length to
210  * 0.
211  * @param vector The vector to clear out.

```

```

206     * @note This will maintain the previous internal capacity of
the vector.
207     */
208 void vectorClear(Vector* vector) {
209     // overwrite internal buffer so every byte is 0
210     memset(vector->internal, 0, vector->internal_size * vector-
>element_size);
211
212     // length = 0 to prevent access to random null chunks?
213     vector->length = 0;
214 }
215
216 /**
217 * Delete a vector, freeing the internal list and zeroing the
struct.
218 * @param vector The vector to delete.
219 */
220 void vectorDelete(Vector* vector) {
221     // free allocated memory block
222     free(vector->internal);
223
224     // prevent further misuse
225     vector->internal = 0;
226     vector->internal_size = 0;
227     vector->element_size = 0;
228     vector->length = 0;
229 }
```

core/vector.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "exception.h"
5
6 typedef struct Vector {
7     void* internal;
8     uint internal_size;
9     uint element_size;
10    uint length;
11 } Vector;
12
13 exception vectorCreate(uint start_capacity, uint element_size,
Vector* vector);
```

```

14 exception vectorAddItem(Vector* vector, const void* item);
15 exception vectorSetItem(const Vector* vector, uint index, const
void* item);
16 exception vectorGetItem(const Vector* vector, uint index, void*
item);
17 exception vectorGetItemPtr(const Vector* vector, uint index,
void** item);
18 exception vectorRemoveItem(Vector* vector, uint index, void*
item);
19 flag vectorPopItem(Vector* vector, void* item);
20 exception vectorInsertItem(Vector* vector, uint index, const
void* item);
21 void vectorClear(Vector* vector);
22 void vectorDelete(Vector* vector);

```

core/yml.c

```

1 #include "yml.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <stdarg.h>
7 #include <errno.h>
8 #include <math.h>
9 #include <unistd.h>
10
11 #include "file.h"
12 #include "list.h"
13 #include "stack.h"
14
15 #define YML_HASHTABLE_SIZE 4
16
17 #define VAR_TOKEN      0b000
18 #define ID_TOKEN       0b001
19 #define LIST_TOKEN     0b010
20 #define IN_TOKEN       0b011
21 #define OUT_TOKEN      0b100
22 #define MODE_MASK      0b111
23
24 #define STRING_TOKEN   0b00001000
25 #define INTEGER_TOKEN  0b00010000
26 #define FLOAT_TOKEN    0b00100000
27 #define BOOL_TOKEN     0b01000000

```

```

28 #define TYPE_MASK      0b01111000
29
30 /**
31  * @brief Represents a word as part of a larger yml file.
32 */
33 typedef struct Word {
34     /** A pointer to the first character in the word. */
35     const char* start;
36
37     /** The number of characters in the word. */
38     uint length;
39
40     /** The indentation of the word in the file - the number
of parents it has. */
41     uint indent;
42
43     /** The line number at which this word occurs in the file.
*/
44     uint line;
45 } Word;
46
47 /**
48  * @brief Represents a token which is an abstract symbol that
can make up a yml structure.
49 */
50 typedef struct Token {
51     /** A pointer to a word that contains the data of the
token. */
52     Word* word;
53
54     /** The type of token */
55     flag type;
56 } Token;
57
58 /**
59  * @brief Allocate memory to copy the content of a yml word
into.
60  * @note This function allocates memory which needs to be
freed.
61  * @param[in] word A pointer to the word that should be
copied.
62  * @param[out] string A pointer to a char pointer where the
address of the new char array should be stored at.

```

```

63  * @throws MEMORY_EXCEPTION if malloc() fails.
64  */
65 exception ymlWordToString(const Word* word, char** string) {
66     *string = (char*)malloc(word->length + 1);
67     if (!(*string)) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for word string.");
68
69     // copy in string data from token
70     memcpy(*string, word->start, word->length);
71
72     // add null terminator character
73     (*string)[word->length] = 0;
74     return SUCCESS;
75 }
76
77 /**
78  * @brief Allocate a new YmlData struct given input data.
79  * @note This function allocates memory which needs to be
freed.
80  * @param[in] param_name The input parameter that will be
copied.
81  * @param[out] yml_data A pointer to a YmlData pointer which
will point to the new struct.
82  * @throws MEMORY_EXCEPTION if malloc() fails.
83  */
84 #define YML_CREATE_FUNC(func_name, func_type, param_name,
yml_type, write_method) \
85 exception func_name(func_type param_name, YmlData** yml_data)
{ \
86     *yml_data = (YmlData*)malloc(sizeof(YmlData)); \
87     if (!(*yml_data)) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for yml data."); \
88     (*yml_data)->type = yml_type; \
89     \
90     /* write_method is the code which is responsible for
setting the YmlData's value */ \
91     /* It is unique for each data type. */ \
92     write_method; \
93     return SUCCESS; \
94 } \
95
96 /**
97  * @brief Allocate a new YmlData struct using a token.

```

```

98     * @note This function allocates memory which needs to be
freed.
99     * @param[in] token A pointer to the token to be used.
100    * @param[out] yml_data A pointer to a YmlData pointer which
will point to the new struct.
101    * @throws MEMORY_EXCEPTION if malloc() fails.
102    * @throws YML_EXCEPTION if converting from a number token
which is outside size range.
103 */
104 #define YML_CREATE_TOKEN_FUNC(func_name, yml_type,
write_method) \
105 exception func_name(const Token* token, YmlData** yml_data)
{ \
106     *yml_data = (YmlData*)malloc(sizeof(YmlData)); \
107     if (!(*yml_data)) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for yml data."); \
108     (*yml_data)->type = yml_type; \
109     \
110     /* This function allocates a string which can be used in
the write_method code to convert to many types */ \
111     char* string; \
112     tekChainThrowThen(ymlWordToString(token->word, &string), { \
\
113         free(*yml_data); \
114     }); \
115     write_method; \
116     return SUCCESS; \
117 } \
118 \
119 /// @copybrief YML_CREATE_FUNC
120 /// @note Create a YmlData struct using a string.
121 YML_CREATE_FUNC(ymlCreateStringData, const char*, string,
STRING_DATA, {
122     const uint len_string = strlen(string) + 1;
123     char* string_copy = (char*)malloc(len_string);
124     if (!string_copy) tekThrowThen(MEMORY_EXCEPTION, "Failed
to allocate memory for string data.", {
125         free(*yml_data);
126     });
127     memcpy(string_copy, string, len_string);
128     (*yml_data)->value = string_copy;
129 });
130

```

```

131 /// @copybrief YML_CREATE_TOKEN_FUNC
132 /// @note Create a YmlData struct using a string token.
133 YML_CREATE_TOKEN_FUNC(ymlCreateStringDataToken, STRING_DATA, {
134     (*yml_data)->value = string;
135 });
136
137 /// @copybrief YML_CREATE_FUNC
138 /// @note Create a YmlData struct using an integer.
139 YML_CREATE_FUNC(ymlCreateIntegerData, const long, integer,
INTEGER_DATA, {
140     (*yml_data)->value = (void*)integer;
141 });
142
143 /// @copybrief YML_CREATE_TOKEN_FUNC
144 /// @note Create a YmlData struct using an integer token.
145 YML_CREATE_TOKEN_FUNC(ymlCreateIntegerDataToken, INTEGER_DATA,
{
146     // convert string into a base-10 number
147     const long integer = strtol(string, NULL, 10);
148     const int error = errno;
149
150     // string is not directly used, just an intermediary to
get the integer value from the token.
151     // therefore it needs to be freed.
152     free(string);
153     if (error == ERANGE) tekThrowThen(YML_EXCEPTION, "Integer
is either too small or large.", {
154         free(*yml_data);
155     });
156     (*yml_data)->value = (void*)integer;
157 });
158
159 /// @copybrief YML_CREATE_FUNC
160 /// @note Create a YmlData struct using a float.
161 YML_CREATE_FUNC(ymlCreateFloatData, const double, number,
FLOAT_DATA, {
162     // memcpy used because you can't cast a double to a void*
163     memcpy(&(*yml_data)->value, &number, sizeof(void*));
164 });
165
166 /// @copybrief YML_CREATE_TOKEN_FUNC
167 /// @note Create a YmlData struct using a float token.
168 YML_CREATE_TOKEN_FUNC(ymlCreateFloatDataToken, FLOAT_DATA, {

```

```

169     const double number = strtod(string, NULL);
170     const int error = errno;
171
172     // free string, as it is no longer needed once the number
has been extracted
173     free(string);
174     if (error == ERANGE) tekThrowThen(YML_EXCEPTION, "Number
is either too small or too large", {
175         free(*yml_data);
176     });
177     memcpy(&(*yml_data)->value, &number, sizeof(void*));
178 });
179
180 /**
181 * @brief A function to automatically allocate a YmlData
struct based on the type of the token.
182 *
183 * This function allocates memory which needs to be freed.
184 *
185 * @param[in] token The token to be used.
186 * @param[out] yml_data A pointer to a YmlData pointer which
will point to the new struct.
187 * @throws MEMORY_EXCEPTION if the subsequent call to allocate
YmlData fails.
188 * @throws YML_EXCEPTION if token data is a number that is
outside the relevant size boundary.
189 * @throws YML_EXCEPTION if token is of an unknown type.
190 */
191 exception ymlCreateAutoDataToken(const Token* token, YmlData** yml_data) {
192     switch (token->type & TYPE_MASK) { // type also contains
other data about the thingy - if it is in a list or not
193         // decide which method to use to create the token.
194         case STRING_TOKEN:
195             tekChainThrow(ymlCreateStringDataToken(token,
yml_data));
196             break;
197         case INTEGER_TOKEN:
198             tekChainThrow(ymlCreateIntegerDataToken(token,
yml_data));
199             break;
200         case FLOAT_TOKEN:

```

```

201             tekChainThrow(ymlCreateFloatDataToken(token,
yml_data));
202         break;
203     default:
204         tekThrow(YML_EXCEPTION, "Bad token type.");
205     }
206     return SUCCESS;
207 }
208
209 /**
210  * @brief Copy the string data of a YmlData struct into a
newly allocated buffer.
211 *
212 * @note This function allocates memory which needs to be
freed.
213 *
214 * @param[in] yml_data A pointer to a YmlData struct to be
used.
215 * @param[out] string A pointer to a string to write the
YmlData into.
216 * @throws YML_EXCEPTION if the YmlData is not a string.
217 * @throws MEMORY_EXCEPTION if malloc() fails.
218 */
219 exception ymlDataToString(const YmlData* yml_data, char**  

string) {
220     // make sure that the data is actually of a string
221     if (yml_data->type != STRING_DATA) tekThrow(YML_EXCEPTION,  

"Data is not of string type.");
222
223     // allocate
224     const char* yml_string = yml_data->value;
225     const uint len_string = strlen(yml_string) + 1;
226     *string = (char*)malloc(len_string * sizeof(char));
227     if (!(*string)) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for string.");
228
229     // copy old string into new buffer
230     memcpy(*string, yml_string, len_string);
231     return SUCCESS;
232 }
233
234 /**

```

```

235  * @brief Copy the integer data of a YmlData struct into an
existing integer.
236  *
237  * @param[in] yml_data A pointer to a YmlData struct to be
used.
238  * @param[out] integer A pointer to a long integer to write
the YmlData into.
239  * @throws YML_EXCEPTION if the YmlData is not an integer.
240  */
241 exception ymlDataToInteger(const YmlData* yml_data, long*
integer) {
242     // make sure that the data is actually of an integer
243     if (yml_data->type != INTEGER_DATA)
tekThrow(YML_EXCEPTION, "Data is not of integer type.");
244     *integer = (long)yml_data->value;
245     return SUCCESS;
246 }
247
248 /**
249 * @brief Copy the double data of a YmlData struct into an
existing double.
250 *
251 * @param[in] yml_data A pointer to a YmlData struct to be
used.
252 * @param[out] number A pointer to a double to write the
YmlData into.
253 * @throws YML_EXCEPTION if the YmlData is not a double.
254 */
255 exception ymlDataToFloat(const YmlData* yml_data, double*
number) {
256     // make sure that the data is actually of a double
257     if (yml_data->type != FLOAT_DATA && yml_data->type !=
INTEGER_DATA) tekThrow(YML_EXCEPTION, "Data is not of double
precision floating point type.");
258     memcpy(number, &yml_data->value, sizeof(void*));
259     return SUCCESS;
260 }
261
262 /**
263 * @brief Copy the data of a YmlData list at a certain index.
264 * @note This function may allocate memory which needs to be
freed. (if using strings)

```

```

265  * @param[in] yml_list A pointer to a YmlData struct to be
used.
266  * @param[in] index The index of the list where the data is
stored.
267  * @param[out] param_name A pointer to a variable to write the
YmlData into.
268  * @throws YML_EXCEPTION if the YmlData is not the correct
type at that index.
269  * @throws MEMORY_EXCEPTION if malloc() fails.
270  */
271 #define YML_GET_LIST_FUNC(func_name, func_type, param_name,
convert_func) \
272 exception func_name(const YmlData* yml_list, const uint index,
func_type param_name) { \
273     if (yml_list->type != LIST_DATA) tekThrow(YML_EXCEPTION,
>Data is not of list type."); \
274     YmlData* yml_data; \
275     tekChainThrow(listGetItem(yml_list->value, index,
&yml_data)); \
276     tekChainThrow(convert_func(yml_data, param_name)); \
277     return SUCCESS; \
278 } \
279 \
280 /// @copydoc YML_GET_LIST_FUNC
281 /// @note Get string from YmlData list. Will allocate new
memory.
282 YML_GET_LIST_FUNC(ymlListGetString, char**, string,
ymlDataToString);
283 \
284 /// @copydoc YML_GET_LIST_FUNC
285 /// @note Get integer from YmlData list.
286 YML_GET_LIST_FUNC(ymlListGetInteger, long*, integer,
ymlDataToInteger);
287 \
288 /// @copydoc YML_GET_LIST_FUNC
289 /// @note Get float from YmlData list.
290 YML_GET_LIST_FUNC(ymlListGetFloat, double*, number,
ymlDataToFloat);
291 \
292 /**
293  * @brief A helper function to fill an existing array by
converting a List of YmlData strings.
294 *

```

```

295  * This function requires an array with the same size as the
list to already be allocated.
296  *
297  * @param[in] yml_list A list containing YmlData structs to be
converted.
298  * @param[out] array An already allocated array of char
pointers.
299  * @throws YML_EXCEPTION if any of the items in the list are
not strings.
300  * @throws MEMORY_EXCEPTION if memory could not be allocated
to store any of the strings.
301  */
302 #define YML_ARRAY_LOOP_FUNC(func_name, func_type,
convert_func) \
303 exception func_name(const List* yml_list, func_type* array)
{ \
304     const ListItem* item = yml_list->data; \
305     uint index = 0; \
306     while (item) { \
307         const YmlData* yml_data = (YmlData*)item->data; \
308         func_type value = 0; \
309         tekChainThrow(convert_func(yml_data, &value)); \
310         array[index++] = value; \
311         item = item->next; \
312     } \
313     return SUCCESS; \
314 } \
315
316 /// @copydoc YML_ARRAY_LOOP_FUNC
317 /// @note Converts a list of YmlData strings into an already
allocated array.
318 YML_ARRAY_LOOP_FUNC(ymlListToStringArrayLoop, char*,
ymlDataToString);
319
320 /// @copydoc YML_ARRAY_LOOP_FUNC
321 /// @note Converts a list of YmlData integers into an already
allocated array.
322 YML_ARRAY_LOOP_FUNC(ymlListToIntegerArrayLoop, long,
ymlDataToInteger);
323
324 /// @copydoc YML_ARRAY_LOOP_FUNC
325 /// @note Converts a list of YmlData integers into an already
allocated array.

```

```

326 YML_ARRAY_LOOP_FUNC(ymlListToFloatArrayLoop, double,
ymlDataToFloat);
327
328 /**
329  * @brief Copy the data of a YmlData list into a newly
allocated array.
330  * @note This function allocated memory which needs to be
freed.
331  * @param[in] yml_list A pointer to a YmlData struct to be
used.
332  * @param[out] array A pointer to store the pointer to the
start of the array.
333  * @param[out] len_array A pointer to an integer to store the
length of the array.
334  * @throws YML_EXCEPTION if the YmlData is not a list.
335  * @throws MEMORY_EXCEPTION if malloc() fails.
336 */
337 #define YML_ARRAY_FUNC(func_name, func_type, loop_func) \
338 exception func_name(const YmlData* yml_list, func_type** \
array, uint* len_array) { \
339     /* Set array length to 0 initially, so if function fails,
potential for loops can't iterate over the non-existent data */ \
340     *len_array = 0; \
341     \
342     /* Non-list data cannot be iterated over by loop_func */ \
343     if (yml_list->type != LIST_DATA) tekThrow(YML_EXCEPTION,
>Data is not of list type."); \
344     \
345     /* Cast the yml_data to a List* so that it can be used by
list functions */ \
346     const List* internal_list = yml_list->value; \
347     \
348     /* Allocate an array that the loop_func can fill */ \
349     *array = (long*)malloc(sizeof(long) * internal_list-
>length); \
350     if (!(*array)) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for array."); \
351     \
352     tekChainThrowThen(loop_func(internal_list, *array), { \
353         free(*array); \
354     }); \
355     \

```

```

356     /* Update length of array to real size if all is well.
*/
357     *len_array = internal_list->length; \
358     return SUCCESS; \
359 } \
360
361 /// @copydoc YML_ARRAY_FUNC
362 /// @note Copy a YmlData list of strings into an array.
Allocates memory that needs to be freed.
363 YML_ARRAY_FUNC(ymlListToStringArray, char*,
ymlListToStringArrayLoop);
364
365 /// @copydoc YML_ARRAY_FUNC
366 /// @note Copy a YmlData list of integers into an array.
Allocates memory that needs to be freed.
367 YML_ARRAY_FUNC(ymlListToIntegerArray, long,
ymlListToIntegerArrayLoop);
368
369 /// @copydoc YML_ARRAY_FUNC
370 /// @note Copy a YmlData list of floats into an array.
Allocates memory that needs to be freed.
371 YML_ARRAY_FUNC(ymlListToFloatArray, double,
ymlListToFloatArrayLoop);
372
373 /**
374  * @brief Allocate a YmlData struct given a list.
375  * @note This function allocates memory which needs to be
freed. The list is only stored as a reference, and not copied. The
list should therefore not be deleted once stored in the YmlData.
376  * @param list A pointer to the list for which the YmlData
should reference.
377  * @param yml_data A pointer to a pointer which should point
to the allocated YmlData.
378  * @throws MEMORY_EXCEPTION if malloc() fails.
379 */
380 exception ymlCreateListData(List* list, YmlData** yml_data) {
381     // allocate some memory for yml data
382     *yml_data = (YmlData*)malloc(sizeof(YmlData));
383     if (!(*yml_data)) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for list data.");
384     (*yml_data)->type = LIST_DATA;
385     (*yml_data)->value = (void*)list;
386     return SUCCESS;

```

```

387 }
388
389 /**
390 * @brief A function to free a YmlData struct based on its
type.
391 * @note This function attempts to free allocated memory: only
YmlData structs created by the other functions should be freed with
this function.
392 * @param yml_data A pointer to the YmlData to free.
393 */
394 void ymlDeleteData(YmlData* yml_data) {
395     if (yml_data->value) {
396         switch (yml_data->type) {
397             case STRING_DATA:
398                 // for string data, we need to deallocate the
copy of the original string
399                 free(yml_data->value);
400                 break;
401             case LIST_DATA:
402                 // for list data, we need to deallocate each
item in the list first
403                 // delete all data before deleting the actual
list.
404                 List* yml_list = yml_data->value;
405                 const ListItem* item = yml_list->data;
406                 while (item) {
407                     YmlData* list_data = item->data;
408                     ymlDeleteData(list_data);
409                     item = item->next;
410                 }
411                 listDelete(yml_list);
412             default:
413                 break;
414             }
415         }
416         free(yml_data);
417     }
418
419 /**
420 * @brief Free a previously allocated YML file.
421 * @note This function should only be called after ymlCreate()
or similar, as it will attempt to free memory.
422 * @param yml A pointer to the YML file to free.

```

```

423 * @throws NULL_PTR_EXCEPTION if yml is NULL
424 * @throws MEMORY_EXCEPTION if there is an error with copying
the yml keys in order to iterate over them and free at each key.
425 */
426 exception ymlDelete(YmlFile* yml) {
427     if (!yml) tekThrow(NULL_PTR_EXCEPTION, "Cannot free null
pointer.");
428
429     // create pointer to store list of values
430     YmlData** values;
431
432     // get number of values
433     const uint num_items = yml->num_items;
434
435     // get values from hashtable
436     tekChainThrow(hashtableGetValues(yml, &values));
437     exception tek_exception = SUCCESS;
438
439     // iterate over the values we got
440     for (uint i = 0; i < num_items; i++) {
441         // if this is a pointer to another hashtable, we need
to recursively free it
442         if (values[i]->type == YML_DATA) {
443             HashTable* hashtable = values[i]->value;
444             tek_exception = ymlDelete(hashtable);
445             tekChainBreak(tek_exception);
446             free(values[i]);
447             // otherwise, just delete as normal
448         } else {
449             ymlDeleteData(values[i]);
450         }
451     }
452 }
453
454 // clean up the array we used
455 free(values);
456 tekChainThrow(tek_exception);
457 return SUCCESS;
458 }
459
460 /**
461 * @brief Instantiate a YmlFile struct.

```

```

462  * @param yml A pointer to an existing but empty YmlFile
463  * struct.
464  */
465 exception ymlCreate(YmlFile* yml) {
466     // simply need to create the hashtable which the yml file
467     // needs to work
468     // this is just the root hash table
469     tekChainThrow(hashtableCreate(yml, YML_HASHTABLE_SIZE));
470     return SUCCESS;
471 }
472 // Auto convert variadic arguments into a list
473 // Using macro to avoid passing around a va_list.
474 /**
475 * Macro template for reading through variadic arguments until
reaching nullptr, and adding each va_arg to a list.
476 * @param list_name The name of the list to create and add
items to.
477 * @param va_name The name of the va_list variable to create.
478 * @param va_type The expected type of the variadic arguments.
479 * @param final_param The name of the final parameter before
the variadic arguments
480 */
481 #define VA_ARGS_TO_LIST(list_name, va_name, va_type,
final_param) \
482 va_list va_name; \
483 va_start(va_name, final_param); \
484 va_type __va_item; \
485 List list_name = {}; \
486 listCreate(&list_name); \
487 while ((__va_item = va_arg(va_name, va_type))) { \
488     listAddItem(&list_name, __va_item); \
489 } \
490 va_end(va_name);
491 /**
493 * @brief Takes a list of keys in order and returns the data
stored under that key.
494 * If we had a YAML file that looked something like this
495 * @code
496 * cpu:
497 *   clock_speed: 3.8GHz

```

```

498 * @endcode
499 * In order to access the CPU's clock speed, we would pass in
a list of:
500 * @code
501 * keys = {"cpu", "clock_speed"}
502 * @endcode
503 * @note This function does NOT copy the data, once yml is
freed, the YmlData is also gone.
504 * @param[in] yml A pointer to the yml file to be read from.
505 * @param[out] data A pointer to an existing but empty YmlData
struct to copy into.
506 * @param[in] keys A list of keys used to access the correct
data.
507 * @throws YML_EXCEPTION If the list points to an invalid key.
508 */
509 exception ymlGetList(YmlFile* yml, YmlData** data, const List*
keys) {
510     *data = 0;
511     HashTable* hashtable = yml;
512     YmlData* loop_data;
513
514     const ListItem* key = keys->data;
515     while (key) {
516         // say we had something like
517         // key: "value"
518         // and i tried to do yml[key][something]
519         // obviously that can't work, so don't allow access if
not a hashtable
520         if (!hashtable)
521             tekThrow(YML_EXCEPTION, "Invalid key -
inaccessible type.");
522
523         // get the data stored in the hashtable, and make sure
that it is a valid key
524         const exception tek_exception =
hashtableGet(hashtable, key->data, &loop_data);
525         if (tek_exception) {
526             tekThrow(YML_EXCEPTION, "Invalid key - key does
not exist.");
527         }
528
529         // if the data stored is type 'yml data', then a
hashtable is stored there

```

```

530         // set this as the new hashtable
531         if (loop_data->type == YML_DATA)
532             hashtable = loop_data->value;
533
534         key = key->next;
535     }
536
537     *data = loop_data;
538     return SUCCESS;
539 }
540
541 /**
542  * Get a piece of yml data by variadic arguments.
543  * @param yml The yml data structure to get from.
544  * @param data The returned data.
545  * @param ... The keys/route to access this data.
546  * @throws YML_EXCEPTION .
547 */
548 exception ymlGetVA(YmlFile* yml, YmlData** data, ...) {
549     // separate method. should be called from a macro ymlGet
550     // the macro will force the final argument to be null
551
552     // start variadic arguments
553     VA_ARGS_TO_LIST(keys_list, keys, const char*, data);
554
555     // get item and throw any exceptions if they occurred
556     const exception tek_exception = ymlGetList(yml, data,
557 &keys_list);
558     listDelete(&keys_list);
559     tekChainThrow(tek_exception);
560     return SUCCESS;
561 }
562
563 /**
564  * @brief Takes a list of keys in order and returns the names
565  * of all keys under that name.
566  * @code
567  * computer:
568  *   cpu:
569  *     manufacturer: intel
570  *     clock_speed: 3.8GHz

```

```

571     *      overclocked: false
572     * @endcode
573     * In order to access keys under cpu, we would use
574     * @code
575     * keys = {"computer", "cpu"}
576     * @endcode
577     * This would return an array of
578     * @code
579     * {"manufacturer", "clock_speed", "overclocked"}
580     * @endcode
581     * @note This function will allocate an array, overwriting the
pointer specified. This array needs to be freed.
582     * @param[in] yml A pointer to the yml file to be read from.
583     * @param[out] yml_keys A pointer to an existing but empty
char* array to copy into.
584     * @param[out] num_keys The number of keys that were written
into the array.
585     * @param[in] keys A list of keys in order to access the
correct data.
586     * @throws YML_EXCEPTION If the list points to an invalid key.
587     */
588 exception ymlGetKeysList(YmlFile* yml, char*** yml_keys, uint*
num_keys, const List* keys) {
589     // get yml data
590     YmlData* yml_data;
591     tekChainThrow(ymlGetList(yml, &yml_data, keys));
592
593     // ensure there are keys to return below this data
594     if (yml_data->type != YML_DATA) tekThrow(YML_EXCEPTION,
"Data has no keys.");
595
596     // return them
597     const HashTable* hashtable = yml_data->value;
598     tekChainThrow(hashtableGetKeys(hashtable, yml_keys));
599     *num_keys = hashtable->num_items;
600     return SUCCESS;
601 }
602
603 /**
604     * Get a list of sub keys from a yml data structure by
variadic arguments.
605     * @param yml The yml data structure to get the data from.

```

```

606  * @param yml_keys The retrieved keys from this data
structure.
607  * @param num_keys The number of keys received
608  * @param ... The keys/route to access this yml data.
609  * @throws YML_EXCEPTION .
610  */
611 exception ymlGetKeysVA(YmlFile* yml, char*** yml_keys, uint*
num_keys, ...) {
612     // separate method. should be called from a macro ymlGet
613     // the macro will force the final argument to be null
614
615     // start variadic arguments
616     VA_ARGS_TO_LIST(keys_list, keys, const char*, num_keys);
617
618     // get item and throw any exceptions if they occurred
619     const exception tek_exception = ymlGetKeysList(yml,
yml_keys, num_keys, &keys_list);
620     listDelete(&keys_list);
621
622     tekChainThrow(tek_exception);
623     return SUCCESS;
624 }
625
626 /**
627 * @brief Takes a list of keys in order and sets the data
stored under that key.
628 * If we had a YAML file that looked something like this
629 * @code
630 * cpu:
631 *   clock_speed: 3.8GHz
632 * @endcode
633 * In order to set the CPU's clock speed, we would pass in a
list of:
634 * @code
635 * keys = {"cpu", "clock_speed"}
636 * @endcode
637 * @param[in] yml A pointer to the yml file to be read from.
638 * @param[out] data A pointer to an existing YmlData struct to
set at the specified key.
639 * @param[in] keys A list of keys used to access the correct
data.
640 * @throws YML_EXCEPTION If the list points to an invalid key.
641 */

```

```

642 exception ymlSetList(YmlFile* yml, YmlData* data, const List*
keys) {
643     // set up some variables
644     const ListItem* key = keys->data;
645
646     HashTable* hashtable = yml;
647     YmlData* loop_data;
648
649     // if the next key is 0, we don't want to process this
650     // hence the use of the separate key and next_key
variables
651     while (key->next) {
652         // get the data stored at current key
653         const exception tek_exception =
hashtableGet(hashtable, key->data, &loop_data);
654
655         // if there is another hash table at that key
656         if (!tek_exception && (loop_data->type == YML_DATA)) {
657             // update searched hash table and increment key
658             hashtable = loop_data->value;
659             key = key->next;
660
661             // go to next iteration
662             continue;
663         }
664
665         // if there isn't a hashtable there, assume that one
should be created
666         HashTable* next_hashtable =
(HashTable*)malloc(sizeof(HashTable));
667         if (!next_hashtable) tekThrow(MEMORY_EXCEPTION,
"Failed to allocate memory for new hash table.");
668
669         // init the hashtable if no memory errors happened
670         tekChainThrowThen(hashtableCreate(next_hashtable,
YML_HASHTABLE_SIZE), {
671             free(next_hashtable);
672         });
673
674         // create a yml data struct to store the hashtable
pointer
675         YmlData* yml_data = (YmlData*)malloc(sizeof(YmlData));

```

```

676             if (!yml_data) tekThrowThen(MEMORY_EXCEPTION, "Failed
to allocate memory for new yml data.", {
677                 hashtableDelete(next_hashtable);
678                 free(next_hashtable);
679             });
680
681             // fill with necessary data if all went well
682             yml_data->type = YML_DATA;
683             yml_data->value = next_hashtable;
684
685             // delete the old data that was stored here, in the
appropriate way
686             if (loop_data && !tek_exception) {
687                 if (loop_data->type == YML_DATA) {
688                     tekChainThrowThen(ymlDelete(loop_data->value),
{
689                         hashtableDelete(next_hashtable);
690                         free(next_hashtable);
691                         free(yml_data);
692                     });
693                 } else {
694                     ymlDeleteData(loop_data);
695                 }
696             }
697
698             // make sure key isn't already there
699             if (hashtableHasKey(hashtable, key->data))
700                 tekThrow(YML_EXCEPTION, "Duplicate key.");
701
702             // set the hashtable at this index
703             tekChainThrowThen(hashtableSet(hashtable, key->data,
yml_data), {
704                 hashtableDelete(next_hashtable);
705                 free(next_hashtable);
706                 free(yml_data);
707             });
708
709             // update hashtable pointer and increment to the next
key
710             hashtable = next_hashtable;
711             key = key->next;
712         }
713

```

```

714     // make sure key isn't already there
715     if (hashtableHasKey(hashtable, key->data))
716         tekThrow(YML_EXCEPTION, "Duplicate key.");
717
718     // check for errors, and then set the data at the correct
key
719     tekChainThrow(hashtableSet(hashtable, key->data, data));
720     return SUCCESS;
721 }
722
723 /**
724 * Set an item in a YML data structure by variadic arguments.
725 * @param yml The yml to set item in.
726 * @param data The yml data to set at that point.
727 * @param ... The keys/route to access that item.
728 * @throws YML_EXCEPTION .
729 */
730 exception ymlSetVA(YmlFile* yml, YmlData* data, ...) {
731     // separate method. should be called from a macro ymlGet
732     // the macro will force the final argument to be null
733
734     VA_ARGS_TO_LIST(keys_list, keys, const char*, data)
735
736     // get item and throw any exceptions if they occurred
737     const exception tek_exception = ymlSetList(yml, data,
&keys_list);
738     listDelete(&keys_list);
739
740     return tek_exception;
741 }
742
743 /**
744 * @brief Takes a list of keys in order and deletes the data
stored under that key.
745 * If we had a YAML file that looked something like this
746 * @code
747 * cpu:
748 *   clock_speed: 3.8GHz
749 * @endcode
750 * In order to delete the CPU's clock speed, we would pass in
a list of:
751 * @code
752 * keys = {"cpu", "clock_speed"}

```

```

753 * @endcode
754 * @param[in] yml A pointer to the yml file to be read from.
755 * @param[in] keys A list of keys used to access the correct
data.
756 * @throws YML_EXCEPTION If the list points to an invalid key.
757 */
758 exception ymlRemoveList(YmlFile* yml, const List* keys) {
759     const ListItem* key = keys->data;
760     const char* prev_key = 0;
761
762     // initial hashtable pointer
763     HashTable* hashtable = yml;
764
765     // place to store data we get during the loop
766     YmlData* loop_data = 0;
767
768     while (key) {
769         // say we had something like
770         // key: "value"
771         // and i tried to do yml[key][something]
772         // obviously that can't work, so don't allow access if
not a hashtable
773         if (!hashtable)
774             tekThrow(YML_EXCEPTION, "Invalid key -
inaccessible type.");
775
776         // get the data stored in the hashtable, and make sure
that it is a valid key
777         const exception tek_exception =
hashtableGet(hashtable, key->data, &loop_data);
778         if (tek_exception)
779             tekThrow(tek_exception, "Invalid key - key does
not exist.");
780
781         // if the data stored is type 'yml data', then a
hashtable is stored there
782         // set this as the new hashtable
783         if (loop_data->type == YML_DATA)
784             hashtable = loop_data->value;
785
786         prev_key = key->data;
787         key = key->next;
788     }

```

```

789
790     // delete the old data that was stored here, in the
appropriate way
791     if (loop_data) {
792         if (loop_data->type == YML_DATA) {
793             tekChainThrow(ymlDelete(loop_data->value));
794         } else {
795             ymlDeleteData(loop_data);
796         }
797     }
798     tekChainThrow(hashtableRemove(hashtable, prev_key));
799     return SUCCESS;
800 }
801
802 /**
803 * Remove item from YML using variadic arguments to specify
the item to remove.
804 * \ref ymlRemove
805 * @param yml The yml file to remove from
806 * @param ... The names of each key to access to arrive at the
item to remove.
807 * @throws YML_EXCEPTION
808 */
809 exception ymlRemoveVA(YmlFile* yml, ...) {
810     // separate method. should be called from a macro
ymlRemove
811     // the macro will force the final argument to be null
812
813     // start variadic arguments
814     VA_ARGS_TO_LIST(keys_list, keys, const char*, yml);
815
816     // get item and throw any exceptions if they occurred
817     const exception tek_exception = ymlRemoveList(yml,
&keys_list);
818     listDelete(&keys_list);
819
820     return tek_exception;
821 }
822
823 /**
824 * @brief Determine if a character is whitespace.
825 * @param[in] c The character to check for whitespace.
826 * @return 1 if whitespace, 0 if not.

```

```

827    */
828 int isWhitespace(const char c) {
829     switch (c) {
830         case 0x00: // null terminator
831         case 0x20: // space
832         case 0x09: // tab
833         case 0x0D: // carriage return
834         case 0x0A: // line feed
835             return 1;
836     default:
837         return 0;
838     }
839 }
840
841 /**
842 * @brief Split a buffer of text into individual words
843 * @param[in] buffer A buffer of text to split into words
844 * @param[in] buffer_size The number of characters in the text
buffer
845 * @param[out] split_text A pointer to an existing list to
append each split Word into.
846 * @throws MEMORY_EXCEPTION If malloc() fails
847 */
848 exception ymlSplitText(const char* buffer, const uint
buffer_size, List* split_text) {
849     int start_index = -1; // marker of where a word starts
850     flag trace_indent = 1; // whether to keep track of the
indent
851     flag inside_marks = 0; // whether this is "inside speech
marks"
852     flag allow_next = 0; // whether this is an escaped
character
853     uint indent = 0; // the number of spaces of indent
854     uint line = 1; // current line number
855     uint last_id_line = 0; // previous line number where id
was found
856     for (int i = 0; i < buffer_size; i++) {
857         // logic to allow for escape characters, if \ skip
next, if skipped next dont skip again
858         if (buffer[i] == '\\') {
859             allow_next = 1;
860             continue;
861         }

```

```

862         if (allow_next) {
863             allow_next = 0;
864             continue;
865         }
866
867         // if speech mark detected, then flip whether inside
marks
868         if (buffer[i] == '"') {
869             inside_marks = (flag)!inside_marks;
870         }
871
872         if (isWhitespace(buffer[i]) || buffer[i] == ':') {
873             // if whitespace is detected outside of speech
marks, a word has ended
874             if ((i > start_index + 1) && !inside_marks) {
875                 Word* word = (Word*)malloc(sizeof(Word));
876                 if (!word) tekThrow(MEMORY_EXCEPTION, "Failed
to allocate memory for word.");
877
878                 word->start = buffer + start_index + 1;
879                 word->length = i - start_index - 1;
880                 word->indent = indent;
881                 word->line = line;
882                 indent = 0;
883                 trace_indent = 0;
884                 tekChainThrow(listAddItem(split_text, word));
885             }
886
887             // if a colon is detected, then an identifier just
ended
888             if (buffer[i] == ':' && !inside_marks) {
889                 if (line == last_id_line)
tekThrow(YML_EXCEPTION, "Cannot have two keys on the same line.")
890
891                 Word* colon = (Word*)malloc(sizeof(Word));
892                 if (!colon) tekThrow(MEMORY_EXCEPTION, "Failed
to allocate memory for word.");
893
894                 colon->start = buffer + i;
895                 colon->length = 1;
896                 colon->indent = 0;
897                 colon->line = line;
898                 indent = 0;

```

```

899             tekChainThrow(listAddItem(split_text, colon));
900             last_id_line = line;
901         }
902
903         if ((buffer[i] == ' ') && (trace_indent)) {
904             indent++;
905         }
906         if (buffer[i] == '\n') {
907             line++;
908             trace_indent = 1;
909         }
910         // when whitespace is detected, restart the
starting position of current word
911         if (!inside_marks) {
912             start_index = i;
913         }
914     }
915 }
916     return SUCCESS;
917 }
918
919 /**
920 * @brief Throw a yml exception, and create a custom message
based on the word and line number.
921 * @param[in] word The word that is responsible for the syntax
error
922 * @throws YML_EXCEPTION Which is the whole point of the
function
923 */
924 exception ymlThrowSyntax(const Word* word) {
925     // utility to create customised error messages
926
927     char error_message[E_MESSAGE_SIZE];
928     sprintf(error_message, "YML syntax error at line %u, in
'", word->line);
929     const uint len_message = strlen(error_message);
930
931     // error messages have limited size, make sure that we
truncate to this size
932     const uint len_copy = E_MESSAGE_SIZE - len_message - 2 >
word->length ? word->length : E_MESSAGE_SIZE - len_message - 2;
933     memcpy(error_message + len_message, word->start,
len_copy);

```

```

934
935     // finish the message with '\0, to complete the opened
quotation mark, and null terminate the string
936     error_message[len_message + len_copy] = '\0';
937     error_message[len_message + len_copy + 1] = 0;
938
939     // no need to mallocate because tekThrow should copy for
us :D
940     tekThrow(YML_EXCEPTION, error_message);
941 }
942
943 /**
944 * @brief Detect the word type
945 * @param[in] word The word to detect its type
946 * @return FLOAT_TOKEN if the word contains digits and a
single decimal point, and potentially starting with '-'
947 * @return INTEGER_TOKEN if the word only contains digits, and
potentially starting with '-'
948 * @return STRING_TOKEN otherwise
949 */
950 flag ymlDetectType(Word* word) {
951     // if the word starts and ends in "", it's a string
952     if ((word->start[0] == '') && (word->start[word->length -
1] == '')) {
953         if (word->length <= 1) return STRING_TOKEN;
954         word->start += 1;
955         word->length -= 2;
956         return STRING_TOKEN;
957     }
958
959     // count letters, numbers and decimal points
960     flag contains_digits = 0;
961     uint num_decimals = 0;
962     flag contains_letters = 0;
963     for (uint i = 0; i < word->length; i++) {
964         const char c = word->start[i];
965         if ((i == 0) && (c == '-')) continue; // allow numbers
to start with a negative sign
966         if (c == '.') num_decimals += 1;
967         else if ((c >= '0') && (c <= '9')) contains_digits =
1;
968         else {
969             contains_letters = 1;

```

```

970             break;
971         }
972         if (num_decimals >= 2) break;
973     }
974
975     // if the word contains anything other than numbers and a
single decimal place, assume string
976     if (contains_letters) return STRING_TOKEN;
977     if (num_decimals > 1) return STRING_TOKEN;
978
979     // based on number of decimals (0 or 1) decides whether
it's an integer or float
980     if ((num_decimals == 1) && contains_digits) return
FLOAT_TOKEN;
981     if ((num_decimals == 0) && contains_digits) return
INTEGER_TOKEN;
982
983     // a single dot with no digits is also a string
984     return STRING_TOKEN;
985 }
986
987 /**
988 * @brief Create a token given a word.
989 * @note This function allocates memory that needs to be
freed.
990 * @param[in] word The word to create a token from.
991 * @param[out] token A pointer to the token that will be
created.
992 */
993 #define TOKEN_CREATE_FUNC(func_name, token_type) \
994 exception func_name(Word* word, Token** token) { \
995     *token = (Token*)malloc(sizeof(Token)); \
996     if (!*token) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for token."); \
997     (*token)->word = word; \
998     (*token)->type = token_type; \
999     return SUCCESS; \
1000 } \
1001
1002 /// @copydoc TOKEN_CREATE_FUNC
1003 /// @note Returns a new token with a type of ID_TOKEN
1004 TOKEN_CREATE_FUNC(ymlCreateKeyToken, ID_TOKEN);
1005

```

```

1006 /// @copydoc TOKEN_CREATE_FUNC
1007 /// @note Returns a new token with an auto-detected type
1008 TOKEN_CREATE_FUNC(ymlCreateValueToken, ymlDetectType(word));
1009
1010 /// @copydoc TOKEN_CREATE_FUNC
1011 /// @note Returns a new token with an auto-detected type +
1012 LIST_TOKEN
1013 TOKEN_CREATE_FUNC(ymlCreateListToken, LIST_TOKEN |
1014 ymlDetectType(word));
1015 /**
1016 * @brief Create an indent token given a type.
1017 * @note This function allocates memory that needs to be
1018 freed.
1019 * @param[in] type The type of indent token to create.
1020 * @param[out] token A pointer to the token that will be
1021 created.
1022 * @throws MEMORY_EXCEPTION if malloc() fails
1023 */
1024 exception ymlCreateIndentToken(const flag type, Token** token)
{
1025     // allocate
1026     *token = (Token*)malloc(sizeof(Token));
1027     if (!*token) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for indent token.");
1028     (*token)->type = type;
1029     (*token)->word = 0; // does not point to a word.
1030     return SUCCESS;
1031 }
1032 /**
1033 * @brief Update the current indent we are at while parsing a
yml file.
1034 * This works not by tracking the number of spaces, but the
number of times that the number of spaces increased since the start
of the file.
1035 * For a file such as
1036 * @code
1037 * depth1:
1038 *   depth2:
1039 *     depth3:
1040 *       depth4: "something"
1041 * @endcode

```

```

1040 * The depth stack would look something like
1041 * @code
1042 * {0, 2, 5, 9}
1043 * @endcode
1044 * Representing the number of spaces at each indent. The size
of the stack - 1 (which is 3 here) represents the current indent of
the file.
1045 * If we updated the file to look like
1046 * @code
1047 * depth1:
1048 *   depth2:
1049 *     depth3:
1050 *       depth4: "something"
1051 *         depth5: "bad formatting"
1052 * @endcode
1053 * The indent stack could be popped repeatedly to find that
the current number of spaces (7) never occurred before, and so this
formatting is invalid.
1054 * @param indent The current indent.
1055 * @param word The word that should be used to check if the
indent has changed.
1056 * @param tokens The list of tokens that should gain an indent
token if the indent has changed.
1057 * @param indent_stack The indent stack - a stack storing each
level of indent in the order that it occurred.
1058 * @throws LIST_EXCEPTION .
1059 * @throws YML_EXCEPTION .
1060 */
1061 exception ymlUpdateIndent(uint* indent, const Word* word,
List* tokens, Stack* indent_stack) {
1062     // essentially, if the indent changes, we need to look at
the indent stack
1063     if (word->indent != *indent) {
1064         Token* indent_token;
1065         // if the indent is more, add this as a new indent to
the stack
1066         if (word->indent > *indent) {
1067             ymlCreateIndentToken(OUT_TOKEN, &indent_token);
1068             stackPush(indent_stack, (void*)*indent);
1069             tekChainThrow(listAddItem(tokens, indent_token));
1070         // otherwise, keep popping until we either find the
indent, or realise its a bad indent
1071     } else if (word->indent < *indent) {

```

```

1072         uint prev_indent = 0;
1073         while (!stackPop(indent_stack, &prev_indent)) {
1074             ymlCreateIndentToken(IN_TOKEN, &indent_token);
1075             tekChainThrow(listAddItem(tokens,
1076                         indent_token));
1077             if (prev_indent == word->indent) {
1078                 break;
1079             }
1080             if (prev_indent < word->indent) {
1081                 tekChainThrow(ymlThrowSyntax(word));
1082             }
1083         }
1084         *indent = word->indent;
1085     }
1086     return SUCCESS;
1087 }
1088
1089 /**
1090  * @brief A function responsible for parsing a Word List, and
1091  * converting this into a series of tokens that can be processed into a
1092  * yml data structure.
1093  * @param split_text A Word List containing a processed yml
1094  * file.
1095  * @param tokens A list of tokens to fill with data.
1096  * @param indent_stack An already existing but empty stack
1097  * that can be used by the function to store the indentation of the
1098  * file.
1099  * @throws YML_EXCEPTION If a syntax error is detected.
1100  * @throws MEMORY_EXCEPTION If malloc() fails.
1101  */
1102 exception ymlCreateTokensStack(const List* split_text, List*
1103 tokens, Stack* indent_stack) {
1104     const ListItem* item = split_text->data;
1105
1106     // start the indent stack at 0 indent
1107     uint indent = 0;
1108     flag expect_value = 0;
1109     tekChainThrow(stackPush(indent_stack, (void*)indent));
1110     while (item) {
1111         Word* word = (Word*)item->data;
1112
1113         // if an unpaired : appears, this is a syntax error

```

```

1108     if ((word->start[0] == ':') && (word->length == 1))
1109         tekChainThrow(ymlThrowSyntax(word));
1110
1111     Token* token = 0;
1112
1113     // if there is a -, this signals a list item
1114     if ((word->start[0] == '-') && (word->length == 1)) {
1115         // there should be a word after this that is the
actual list data
1116         if ((!item->next) || (!expect_value))
1117             tekChainThrow(ymlThrowSyntax(word));
1118         item = item->next;
1119         Word* next_word = (Word*)item->data;
1120         tekChainThrow(ymlCreateListToken(next_word,
&token));
1121     } else {
1122         // last item should always be a value, an
identifier cant identify nothing
1123         if (!item->next) {
1124             tekChainThrow(ymlCreateValueToken(word,
&token));
1125         } else {
1126             const Word* next_word = (Word*)item->next-
>data;
1127             // if next word is ':', this is an identifier
1128             if ((next_word->start[0] == ':') &&
(next_word->length == 1)) {
1129                 tekChainThrow(ymlCreateKeyToken(word,
&token));
1130                 const uint prev_indent = indent;
1131                 tekChainThrow(ymlUpdateIndent(&indent,
word, tokens, indent_stack));
1132
1133                 if ((indent > prev_indent) && !
expect_value) {
1134                     tekChainThrow(ymlThrowSyntax(word));
1135                 }
1136
1137                 // signal that we expect to find data
after this
1138                 expect_value = 1;
1139
1140                 // skip over the ':', not needed data

```

```

1141             item = item->next;
1142             // otherwise, its a value
1143         } else {
1144             if (!expect_value)
1145                 tekChainThrow(ymlThrowSyntax(word));
1146             tekChainThrow(ymlCreateValueToken(word,
1147 &token));
1148             // we do not expect to find multiple
1149             values in a row, unless a list
1150             expect_value = 0;
1151         }
1152     }
1153
1154     tekChainThrow(listAddItem(tokens, token));
1155     item = item->next;
1156 }
1157 return SUCCESS;
1158 }
1159
1160 /**
1161 * @copybrief ymlCreateTokensStack
1162 * @param split_text A Word List containing a processed yml
1163 file.
1164 * @param tokens A list of tokens to fill with data.
1165 * @throws YML_EXCEPTION If there is a syntax error
1166 * @throws MEMORY_EXCEPTION If malloc() fails.
1167 */
1168 exception ymlCreateTokens(const List* split_text, List*
tokens) {
1169     Stack indent_stack;
1170     stackCreate(&indent_stack);
1171
1172     // ymlCreateTokensStack operates with a while loop.
1173     // Creating the indent stack outside of the function
1174     // allows us to return at any point in the loop, and not worry about
1175     // freeing the stack.
1176     // The stack creation and freeing is all this function
1177     // does.
1178     const exception tek_exception =
1179     ymlCreateTokensStack(split_text, tokens, &indent_stack);
1180     stackDelete(&indent_stack);

```

```

1176     return tek_exception;
1177 }
1178
1179 /**
1180  * @brief Helper function to turn a list of tokens into a
1181  * single YmlData struct.
1182  * @note This function allocates memory that needs to be
1183  * freed.
1184  * @param prev_item A pointer to prev_item in main function.
1185  * @param item A pointer to item in main function.
1186  * @param yml_list A list of YmlData that will be used in the
1187  * creation of the YmlData.
1188  * @param yml_data A pointer to the YmlData.
1189  * @throws YML_EXCEPTION if a syntax error is detected.
1190  * @throws MEMORY_EXCEPTION if malloc() fails.
1191 */
1192 exception ymlFromTokensListAddList(const ListItem** prev_item,
1193 const ListItem** item, List* yml_list, YmlData** yml_data) {
1194     // list of tokens will be like:
1195     // ... data ...
1196     // list identifier: <- prev item
1197     //     item 1           <- item
1198     //     item 2
1199     // NON-LIST-ITEM ...
1200     // so search for the first non-list item
1201     while (*item) {
1202         // loop until we find a non list item in the tokens
1203         const Token* list_token = (Token*)(*item)->data;
1204         if ((list_token->type & MODE_MASK) != LIST_TOKEN)
1205             break;
1206
1207         // attempt to fill list with yml data from tokens.
1208         YmlData* list_data;
1209         tekChainThrow(ymlCreateAutoDataToken(list_token,
1210             &list_data));
1211         tekChainThrowThen(listAddItem(yml_list, list_data), {
1212             ymlDeleteData(list_data);
1213         });
1214
1215         *prev_item = *item;
1216         *item = (*item)->next;
1217     }
1218

```

```

1213     // create the actual yml data now
1214     tekChainThrowThen(ymlCreateListData(yml_list, yml_data), {
1215         ymlDeleteData(*yml_data);
1216     });
1217
1218     return SUCCESS;
1219 }
1220
1221 /**
1222  * @brief A function to read a series of tokens into YmlData
1223  * structs.
1224  * @param tokens A list of tokens to be converted.
1225  * @param yml A pointer to the yml file.
1226  * @param keys_list A preallocated, empty list that can be
1227  * used to store the keys needed to access the yml file.
1228  * @throws YML_EXCEPTION if a syntax error is detected.
1229  * @throws MEMORY_EXCEPTION if malloc() fails.
1230 */
1231 exception ymlFromTokensList(const List* tokens, YmlFile* yml,
1232 List* keys_list) {
1233     // when an ID is encountered, the last item is swapped
1234     // with the new id
1235     // therefore, this initial null value is needed to allow
1236     // for this swap
1237     listAddItem(keys_list, 0);
1238     char* key = 0;
1239     //exception tek_exception = SUCCESS;
1240
1241     const ListItem* item = tokens->data;
1242     while (item) {
1243         const Token* token = item->data;
1244
1245         // using ymlSetList requires a list of keys, in the
1246         // order of access, so:
1247         // material:
1248         //   color: "red"
1249         // is accessed with the list ["material", "color"]
1250
1251         // to begin, we should check for control tokens
1252         // if there is an ID, swap out the last item
1253         if (token->type == ID_TOKEN) {
1254             tekChainThrow(listPopItem(keys_list, &key));
1255             if (key) free(key);

```

```

1250         tekChainThrow(ymlWordToString(token->word, &key));
1251         tekChainThrow(listAddItem(keys_list, key));
1252         item = item->next;
1253         continue;
1254     }
1255     // if going "out", or extending the depth of the data,
add a null to signify another layer of depth
1256     if (token->type == OUT_TOKEN) {
1257         tekChainThrow(listAddItem(keys_list, 0));
1258         item = item->next;
1259         continue;
1260     }
1261     // if going "in", or reducing the depth, pop the item
to signify a smaller depth
1262     if (token->type == IN_TOKEN) {
1263         tekChainThrow(listPopItem(keys_list, &key));
1264         if (key) free(key);
1265         item = item->next;
1266         continue;
1267     }
1268     // for any control token, we should skip to processing
the next item
1269     // no further processing needs to be done for them
1270
1271     // now start creating the yml data to write at this
location
1272     YmlData* yml_data;
1273
1274     // if the token is a list, we could be adding several
values under the same key
1275     // this could require going through several items in
the token list
1276     if ((token->type & LIST_TOKEN) == LIST_TOKEN) {
1277         List* yml_list = (List*)malloc(sizeof(List));
1278         if (!yml_list) tekThrow(MEMORY_EXCEPTION, "Failed
to allocate memory for yml list.");
1279         listCreate(yml_list);
1280
1281         // store a pointer to prev_item
1282         // if we iterate to the next item, and it is not
part of the list, then the inner loop will end
1283         // however, the outer loop will always iterate to
the next item, skipping this item

```

```

1284          // for this reason, we need to try and leave the
1285          iterator in a good position for the loop
1286          const ListItem* prev_item = item;
1287
1288          // use external function to add all items of list
1289          // if it fails, then we need to free the data it
1290          removed, otherwise we will lose pointer to it.
1291
1292          tekChainThrowThen(ymlFromTokensListAddList(&prev_item, &item,
1293          yml_list, &yml_data), {
1294              const ListItem* del_item = yml_list->data;
1295              while (del_item) {
1296                  YmlData* del_data = del_item->data;
1297                  ymlDeleteData(del_data);
1298                  del_item = del_item->next;
1299              }
1300              listDelete(yml_list);
1301              free(yml_list);
1302          });
1303          item = prev_item;
1304
1305          // non list tokens should just be added normally
1306      } else {
1307          tekChainThrow(ymlCreateAutoDataToken(token,
1308          &yml_data));
1309      }
1310
1311      tekChainThrow(ymlSetList(yml, yml_data, keys_list));
1312      item = item->next;
1313  }
1314
1315 /**
1316 * @copybrief ymlFromTokensList
1317 * @param tokens
1318 * @param yml
1319 * @throws YML_EXCEPTION if a syntax error is detected.

```

```

1320 * @throws MEMORY_EXCEPTION if malloc() fails.
1321 */
1322 exception ymlFromTokens(const List* tokens, YmlFile* yml) {
1323     List keys_list = {};
1324     listCreate(&keys_list);
1325
1326     // use pre-made list to run helper function
1327     // this means the list can be properly freed if something
goes wrong, even if we return from the while loop early.
1328     const exception tek_exception = ymlFromTokensList(tokens,
yml, &keys_list);
1329     if (tek_exception) {
1330         listFreeAllData(&keys_list);
1331         listDelete(&keys_list);
1332     }
1333
1334     return tek_exception;
1335 }
1336
1337 /**
1338 * @brief Format and print YmlData
1339 * @note This function will print with a newline character at
the end.
1340 * @param data The YmlData to print
1341 */
1342 void ymlPrintData(const YmlData* data) {
1343     // these two are easy
1344     if (data->type == STRING_DATA) printf("%s\n",
(char*)data->value);
1345     if (data->type == INTEGER_DATA) printf("%ld\n",
(long)data->value);
1346     if (data->type == FLOAT_DATA) {
1347         // cannot cast a void* to double, so need to memcpy
1348         double number = 0;
1349         memcpy(&number, &data->value, sizeof(void*));
1350         printf("%f\n", number);
1351     }
1352 }
1353
1354 /**
1355 * @brief A helper function to be called recursively, tracking
the current indentation that we are printing at.
1356 * @param yml The yml file to be printed.

```

```

1357 * @param indent The current indentation.
1358 * @throws MEMORY_EXCEPTION If failed to allocate memory for
keys/values to print.
1359 */
1360 exception ymlPrintIndent(YmlFile* yml, uint indent) {
1361     if (!yml) tekThrow(NULL_PTR_EXCEPTION, "Cannot print null
pointer.");
1362
1363     const char** keys;
1364     tekChainThrow(hashtableGetKeys(yml, &keys));
1365
1366     YmlData** values;
1367     tekChainThrow(hashtableGetValues(yml, &values));
1368
1369     const uint num_items = yml->num_items;
1370     exception tek_exception = SUCCESS;
1371
1372     // iterate over the values we got
1373     for (uint i = 0; i < num_items; i++) {
1374         for (uint j = 0; j < indent; j++) printf(" ");
1375         printf("%s: ", keys[i]);
1376         // if this is a pointer to another hashtable, we need
to recursively print it
1377         if (values[i]->type == YML_DATA) {
1378             HashTable* hashtable = values[i]->value;
1379             printf("\n");
1380             tek_exception = ymlPrintIndent(hashtable, indent +
1);
1381             if (tek_exception) break;
1382             // otherwise, just print as normal
1383         } else {
1384             // if a list, print each item separately
1385             if (values[i]->type == LIST_DATA) {
1386                 List* yml_list = values[i]->value;
1387                 ListItem* item = yml_list->data;
1388                 printf("\n");
1389                 while (item) {
1390                     const YmlData* list_data = item->data;
1391                     // format so that it sits underneath the
key, indented correctly
1392                     for (uint i = 0; i <= indent; i++)
printf(" ");
1393                     printf("- ");

```

```

1394                     ymlPrintData(list_data);
1395                     item = item->next;
1396                 }
1397             } else {
1398                 ymlPrintData(values[i]);
1399             }
1400         }
1401     }
1402
1403     // clean up the array we used
1404     free(keys);
1405     free(values);
1406     tekChainThrow(tek_exception);
1407     return SUCCESS;
1408 }
1409
1410 /**
1411 * @brief Formats and prints yml file.
1412 * @param yml The yml file to print
1413 * @throws MEMORY_EXCEPTION if failed to allocate memory for
keys/values to print
1414 */
1415 exception ymlPrint(YmlFile* yml) {
1416     tekChainThrow(ymlPrintIndent(yml, 0)); // recursive func,
initial indent = 0
1417     return SUCCESS;
1418 }
1419
1420 /**
1421 * @brief Take a filename and read its content into a yml data
structure.
1422 * @param filename The filename of the yml file to be read.
1423 * @param yml A pointer to an existing but empty YmlFile
struct to read the data into.
1424 * @throws YML_EXCEPTION if could not read file.
1425 */
1426 exception ymlReadFile(const char* filename, YmlFile* yml) {
1427     tekChainThrow(ymlCreate(yml));
1428
1429     // allocate enough memory and read the file into it
1430     uint file_size;
1431     tekChainThrow(getFileSize(filename, &file_size))
1432     char* buffer = (char*)malloc(file_size * sizeof(char)));

```

```

1433     if (!buffer) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory to read yml file into.");
1434     tekChainThrow(readFile(filename, file_size, buffer));
1435
1436     // create list to store split up text.
1437     List split_text;
1438     listCreate(&split_text);
1439
1440     // split the text into atomic units, the supposed keys and
values
1441     tekChainThrowThen(ymlSplitText(buffer, file_size,
&split_text), {
1442         listFreeAllData(&split_text);
1443         listDelete(&split_text);
1444         free(buffer);
1445     });
1446
1447     List tokens;
1448     listCreate(&tokens);
1449
1450     // check the split text and convert to useable tokens
1451     tekChainThrowThen(ymlCreateTokens(&split_text, &tokens), {
1452         listFreeAllData(&split_text);
1453         listFreeAllData(&tokens);
1454         listDelete(&split_text);
1455         listDelete(&tokens);
1456         free(buffer);
1457     });
1458
1459     // build yml data from tokens
1460     tekChainThrowThen(ymlFromTokens(&tokens, yml), {
1461         listFreeAllData(&split_text);
1462         listFreeAllData(&tokens);
1463         listDelete(&split_text);
1464         listDelete(&tokens);
1465         ymlDelete(yml);
1466         free(buffer);
1467     });
1468
1469     // clean up
1470     listFreeAllData(&split_text);
1471     listFreeAllData(&tokens);
1472     listDelete(&split_text);

```

```

1473     listDelete(&tokens);
1474     free(buffer);
1475
1476     return SUCCESS;
1477 }

```

core/yml.h

```

1  #pragma once
2
3  #define YML_DATA      0
4  #define STRING_DATA   1
5  #define INTEGER_DATA  2
6  #define FLOAT_DATA    3
7  #define LIST_DATA     4
8
9  #include "../tekgl.h"
10 #include "../core/exception.h"
11 #include "../core/hashtable.h"
12
13 typedef struct YmlData {
14     flag type;
15     void* value;
16 } YmlData;
17
18 typedef HashTable YmlFile;
19
20 exception ymlCreate(YmlFile* yml);
21 exception ymlReadFile(const char* filename, YmlFile* yml);
22
23 exception ymlGetVA(YmlFile* yml, YmlData** data, ...);
24 /// @copydoc ymlGetVA
25 #define ymlGet(yml, data, ...) ymlGetVA(yml, data, __VA_ARGS__, 0)
26
27 exception ymlGetKeysVA(YmlFile* yml, char*** yml_keys, uint* num_keys, ...);
28 /// @copydoc ymlGetKeysVA
29 #define ymlGetKeys(yml, data, num_keys, ...) ymlGetKeysVA(yml, data, num_keys, __VA_ARGS__, 0);
30
31 exception ymlDataToString(const YmlData* yml_data, char** string);

```

```

32 exception ymlDataToInteger(const YmlData* yml_data, long*
integer);
33 exception ymlDataToFloat(const YmlData* yml_data, double*
number);
34
35 exception ymlListGetString(const YmlData* yml_list, uint index,
char** string);
36 exception ymlListGetInteger(const YmlData* yml_list, uint index,
long* integer);
37 exception ymlListGetFloat(const YmlData* yml_list, uint index,
double* number);
38
39 exception ymlListToStringArray(const YmlData* yml_list, char*** array,
uint* len_array);
40 exception ymlListToIntegerArray(const YmlData* yml_list, long** array,
uint* len_array);
41 exception ymlListToFloatArray(const YmlData* yml_list, double** array,
uint* len_array);
42
43 exception ymlSetVA(YmlFile* yml, YmlData* data, ...);
44 #define ymlSet(yml, data, ...) ymlSetVA(yml, data, __VA_ARGS__, 0)
45
46 exception ymlRemoveVA(YmlFile* yml, ...);
47 #define ymlRemove(yml, ...) ymlRemoveVA(yml, __VA_ARGS__, 0)
48
49 exception ymlDelete(YmlFile* yml);
50
51 exception ymlPrint(YmlFile* yml);

```

tekgl/camera.c

```

1 #include "camera.h"
2
3 #include <stdio.h>
4 #include <string.h>
5 #include <cglm/affine.h>
6 #include "manager.h"
7 #include "../core/list.h"
8 #include <cglm/cam.h>
9
10 static List cameras = {0, 0};
11 static float aspect_ratio = 1.0f;
12

```

```

13 /**
14  * Update the projection matrix of the camera.
15  * @param camera A pointer to the camera to update.
16 */
17 static void tekUpdateCameraProjection(TekCamera* camera) {
18     // glm provides method to create perspective matrix.
19     glm_perspective(camera->fov, aspect_ratio, camera->near,
camera->far, camera->projection);
20 }
21
22 /**
23  * Update the view matrix of the camera.
24  * @param camera A pointer to the camera to update.
25 */
26 static void tekUpdateCameraView(TekCamera* camera) {
27     const double pitch = (double)camera->rotation[1], yaw =
(double)camera->rotation[0];
28     vec3 look_at = {};
29     vec3 up = { 0.0f, 1.0f, 0.0f };
30
31     // create a vector based on pitch and yaw, that points in
the same direction as the camera.
32     look_at[0] = (float)(cos(yaw) * cos(pitch)) + camera-
>position[0];
33     look_at[1] = (float)sin(pitch) + camera->position[1];
34     look_at[2] = (float)(sin(yaw) * cos(pitch)) + camera-
>position[2];
35
36     // then, look at this point. so we look in the same way as
expected.
37     glm_lookat(camera->position, look_at, up, camera->view);
38 }
39
40 /**
41  * Called when the framebuffer size changes, e.g. when the user
resizes the window.
42  * @param framebuffer_width The new width of the framebuffer.
43  * @param framebuffer_height The new height of the framebuffer.
44 */
45 static void tekCameraFramebufferCallback(const int
framebuffer_width, const int framebuffer_height) {
46     // calculate aspect ratio

```

```

47     aspect_ratio = (float)framebuffer_width /
48     (float)framebuffer_height;
49     // update every camera with the new aspect ratio.
50     const ListItem* item;
51     foreach(item, (&cameras), {
52         TekCamera* camera = (TekCamera*)item->data;
53         tekUpdateCameraProjection(camera);
54     });
55 }
56
57 /**
58 * Called when program exits, cleans up some structures.
59 */
60 static void tekCameraDeleteFunc() {
61     listDelete(&cameras); // delete list of cameras.
62 }
63
64 /**
65 * Initialise some structures needed to have cameras.
66 */
67 tek_init tekCameraInit() {
68     // framebuffer callback to wait for when viewport changes
69     tekAddFramebufferCallback(tekCameraFramebufferCallback);
70
71     // call delete func when the program exits
72     tekAddDeleteFunc(tekCameraDeleteFunc);
73
74     // create empty list of cameras.
75     listCreate(&cameras);
76 }
77
78 /**
79 * Update the position of a camera and update any matrices this
80 would affect.
81 * @param camera The camera to update.
82 * @param position The new position of the camera.
83 */
84 void tekSetCameraPosition(TekCamera* camera, vec3 position) {
85     memcpy(camera->position, position, sizeof(vec3)); // copy
86     in position
87     tekUpdateCameraView(camera); // update matrices.
88 }
```

```

87
88  /**
89   * Set the rotation of the camera and update the matrices it
90   * affects.
91   * @param camera The camera to update.
92   * @param rotation The new rotation of the camera.
93   */
94 void tekSetCameraRotation(TekCamera* camera, vec3 rotation) {
95     memcpy(camera->rotation, rotation, sizeof(vec3)); // copy
96     rotation in
97     tekUpdateCameraView(camera); // update
98 }
99 /**
100  * Create a camera struct, a container for some information
101  * about a camera + matrices needed to render with this camera.
102  * @param camera A pointer to an empty TekCamera struct.
103  * @param position The position of the camera.
104  * @param rotation The rotation of the camera.
105  * @param fov The fov (field of view) of the camera in radians.
106  * @param near The near clipping plane of the camera. (things
107  * closer than this will be invisible)
108  * @param far The far clipping plane of the camera. (things
109  * further than this will be invisible)
110  * @throws LIST_EXCEPTION if could not add camera to list of
111  * cameras.
112  */
113 exception tekCreateCamera(TekCamera* camera, vec3 position,
114                           vec3 rotation, const float fov, const float near, const float far) {
115     // when window is resized, need to update cameras to match
116     // new aspect ratio
117     // so all cameras need to be recorded to be updated when
118     // needed.
119     tekChainThrow(listAddItem(&cameras, camera));
120     // generate matrices.

```

```
121     tekUpdateCameraView(camera);
122     tekUpdateCameraProjection(camera);
123     return SUCCESS;
124 }
```

tekgl/camera.h

```
1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../core/exception.h"
5
6 #include <cglm/vec3.h>
7 #include <cglm/mat4.h>
8
9 typedef struct TekCamera {
10     vec3 position;
11     vec3 rotation;
12     mat4 view;
13     mat4 projection;
14     float fov;
15     float near;
16     float far;
17 } TekCamera;
18
19 exception tekCreateCamera(TekCamera* camera, vec3 position, vec3
rotation, float fov, float near, float far);
20 void tekSetCameraPosition(TekCamera* camera, vec3 position);
21 void tekSetCameraRotation(TekCamera* camera, vec3 rotation);
```

tekgl/entity.c

```
1 #include "entity.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 #include "../core/hashtable.h"
8
9 static flag mesh_cache_init = 0, material_cache_init = 0;
10 static HashTable mesh_cache = {}, material_cache = {};
11 static TekMaterial* using_material = 0;
12
13 /**
```

```

14  * The cleanup function for the entity code. Deletes caches
uses for materials and meshes.
15  */
16 static void tekEntityDelete() {
17     // loop through each mesh and delete it.
18     TekMesh** meshes;
19     if (hashtableGetValues(&mesh_cache, &meshes) == SUCCESS) {
20         for (uint i = 0; i < mesh_cache.num_items; i++) {
21             tekDeleteMesh(meshes[i]);
22             free(meshes[i]);
23         }
24     }
25     // do the same for the materials
26     TekMaterial** materials;
27     if (hashtableGetValues(&material_cache, &materials) ==
SUCCESS) {
28         for (uint i = 0; i < material_cache.num_items; i++) {
29             tekDeleteMaterial(materials[i]);
30             free(materials[i]);
31         }
32     }
33     // once everything is deleted, can now free the hashtables.
34     if (mesh_cache_init) hashtableDelete(&mesh_cache);
35     if (material_cache_init) hashtableDelete(&material_cache);
36 }
37
38 /**
39  * Initialise some surrounding helper structures for entities,
notably caches for meshes and materials.
40 */
41 tek_init tekEntityInit(void) {
42     // caches
43     if (hashtableCreate(&mesh_cache, 4) == SUCCESS)
mesh_cache_init = 1;
44     if (hashtableCreate(&material_cache, 4) == SUCCESS)
material_cache_init = 1;
45
46     // specify the cleanup function
47     tekAddDeleteFunc(tekEntityDelete);
48 }
49
50 /**

```

```

51  * A template function for creating/caching things loaded from
files. Used to make a cache of materials and meshes.
52  * @param func_name The name of the function
53  * @param func_type The type of the output for this function.
54  * @param param_name The name of the parameter that is
outputted
55  * @param create_func The function that will create
"func_type"s.
56  * @param delete_func The function that will delete
"func_type"s if something goes wrong.
57  */
58 #define REQUEST_FUNC(func_name, func_type, param_name,
create_func, delete_func) \
59 static exception func_name(const char* filename, func_type** \
param_name) { \
60     if (!param_name##_cache_init) tekThrow(NULL_PTR_EXCEPTION, \
"Cache does not exist."); \
61     if (hashtableHasKey(&param_name##_cache, filename)) { \
62         tekChainThrow(hashtableGet(&param_name##_cache, \
filename, param_name)); \
63     } else { \
64         *param_name = (func_type*)malloc(sizeof(func_type)); \
65         if (!(*param_name)) tekThrow(MEMORY_EXCEPTION, "Failed \
to allocate memory for cache."); \
66         tekChainThrow(create_func(filename, *param_name)); \
67         tekChainThrowThen(hashtableSet(&param_name##_cache, \
filename, *param_name), { \
68             delete_func(*param_name); \
69             free(*param_name); \
70         }); \
71     } \
72     return SUCCESS; \
73 } \
74 \
75 REQUEST_FUNC(requestMesh, TekMesh, mesh, tekReadMesh, \
tekDeleteMesh);
76 REQUEST_FUNC(requestMaterial, TekMaterial, material, \
tekCreateMaterial, tekDeleteMaterial);
77 /**
78 /**
79 * Create a new entity from a mesh and material file, along
with other data.
80 * mesh file type = .tmsh, material file type = .tmat

```

```

81     * @param mesh_filename The mesh file to use for this entity.
82     * @param material_filename The material file to use for this
entity.
83     * @param position The position of the entity.
84     * @param rotation The rotation of the entity.
85     * @param scale The scale of the entity.
86     * @param entity A pointer to an existing entity struct that
will have entity data written into it.
87     * @throws MEMORY_EXCEPTION if malloc() fails.
88     */
89 exception tekCreateEntity(const char* mesh_filename, const
char* material_filename, vec3 position, vec4 rotation, vec3 scale,
TekEntity* entity) {
90     // check if mesh has been created already
91     // if yes, retreive from cache, otherwise create new one.
92     TekMesh* mesh;
93     tekChainThrow(requestMesh(mesh_filename, &mesh));
94
95     // do the same for the material
96     TekMaterial* material;
97     tekChainThrow(requestMaterial(material_filename,
&material));
98
99     // write data into entity struct.
100    entity->mesh = mesh;
101    entity->material = material;
102    glm_vec3_copy(position, entity->position);
103    glm_vec4_copy(rotation, entity->rotation);
104    glm_vec3_copy(scale, entity->scale);
105
106    return SUCCESS;
107 }
108
109 /**
110  * Update an entity with a new position and rotation.
111  * @param entity The entity to update.
112  * @param position The new position of the entity.
113  * @param rotation The new rotation of the entity.
114  */
115 void tekUpdateEntity(TekEntity* entity, vec3 position, vec4
rotation) {
116     glm_vec3_copy(position, entity->position); // update
position

```

```

117     glm_vec4_copy(rotation, entity->rotation); // update
rotation
118 }
119
120 /**
121 * Draw an entity to the screen from the perspective of a
camera.
122 * Also use the material associated with the entity.
123 * @param entity The entity to draw.
124 * @param camera The camera which contains the perspective to
draw from.
125 * @throws SHADER_EXCEPTION if could not set shader uniforms
for camera.
126 */
127 exception tekDrawEntity(TekEntity* entity, TekCamera* camera) {
128     // use entity's material
129     tekChainThrow(tekBindMaterial(entity->material));
130
131     // create the model matrix for this entity
132     // this specifies how the model is moved from its initial
position
133     mat4 translation;
134     glm_translate_make(translation, entity->position);
135     mat4 rotation;
136     glm_quat_mat4(entity->rotation, rotation);
137     mat4 scale;
138     glm_scale_make(scale, entity->scale);
139     mat4 model;
140     glm_mat4_mul(rotation, scale, model);
141     glm_mat4_mul(translation, model, model);
142
143     // load all the matrices into the shader
144     // camera contains most of them already.
145     if (tekMaterialHasUniformType(entity->material,
MODEL_MATRIX_DATA))
146         tekChainThrow(tekBindMaterialMatrix(entity->material,
model, MODEL_MATRIX_DATA));
147
148     if (tekMaterialHasUniformType(entity->material,
VIEW_MATRIX_DATA))
149         tekChainThrow(tekBindMaterialMatrix(entity->material,
camera->view, VIEW_MATRIX_DATA));
150

```

```

151     if (tekMaterialHasUniformType(entity->material,
PROJECTION_MATRIX_DATA))
152         tekChainThrow(tekBindMaterialMatrix(entity->material,
camera->projection, PROJECTION_MATRIX_DATA));
153
154     if (tekMaterialHasUniformType(entity->material,
CAMERA_POSITION_DATA))
155         tekChainThrow(tekBindMaterialVec3(entity->material,
camera->position, CAMERA_POSITION_DATA));
156
157     // draw to the screen
158     tekDrawMesh(entity->mesh);
159
160     return SUCCESS;
161 }
162
163 /**
164 * Notify the materials renderer that the material has been
changed without using the material code directly.
165 */
166 void tekNotifyEntityMaterialChange() {
167     using_material = 0; // bodgemethod
168     // i did this to debug and then it worked fine so its
become permanent.
169 }
```

tekgl/entity.h

```

1 #pragma once
2
3 #include <cglm/vec3.h>
4 #include <cglm/quat.h>
5
6 #include "manager.h"
7 #include "mesh.h"
8 #include "material.h"
9 #include "camera.h"
10
11 typedef struct TekEntity {
12     TekMesh* mesh;
13     TekMaterial* material;
14     vec3 position;
15     vec4 rotation;
16     vec3 scale;
```

```

17 } TekEntity;
18
19 exception tekCreateEntity(const char* mesh_filename, const char*
material_filename, vec3 position, vec4 rotation, vec3 scale,
TekEntity* entity);
20 void tekUpdateEntity(TekEntity* entity, vec3 position, vec4
rotation);
21 exception tekDrawEntity(TekEntity* entity, TekCamera* camera);
22 void tekNotifyEntityMaterialChange();

```

tekgl/font.c

```

1 #include "font.h"
2
3 #include <math.h>
4 #include <freetype/freetype.h>
5
6 #include "glad/glad.h"
7 #define FT_FREETYPE_H
8
9 #define ATLAS_QUIT      0b1
10 #define ATLAS_QUIT_ALREADY 0b10
11 #define ATLAS_FIRST_TRIAL 0b100
12 #define ATLAS_MIN_SIZE    2
13
14 FT_Library ft_library = 0;
15
16 /**
17  * Initialise the freetype library and allow fonts to be loaded
and used.
18  * @throws FREETYPE_EXCEPTION if library could not be
initialised.
19 */
20 exception tekCreateFreeType() {
21     // initialise freetype library if it hasn't been already
22     if (!ft_library)
23         if (FT_Init_FreeType(&ft_library))
tekThrow(FREETYPE_EXCEPTION, "Failed to initialise FreeType.");
24     return SUCCESS;
25 }
26
27 /**
28  * Delete the freetype library once font loading is finished.
29 */

```

```

30 void tekDeleteFreeType() {
31     if (ft_library) FT_Done_FreeType(ft_library); // allows to
access static variable from outside
32 }
33
34 /**
35 * Create a font face from a filename and a face index. Each
font file has different faces e.g. bold, italic.
36 * The face index specifies which one of these faces you want.
37 * @param filename The name of the font file to load.
38 * @param face_index The face index to load. (use 0 if unsure)
39 * @param face_size The size of the face in pixels to load as.
40 * @param face The outputted font face.
41 * @throws FREETYPE_EXCEPTION if the font could not be created.
42 */
43 exception tekCreateFontFace(const char* filename, const uint
face_index, const uint face_size, FT_Face* face) {
44     // make sure that the font is large enough
45     if (face_size < MIN_FONT_SIZE) tekThrow(FREETYPE_EXCEPTION,
"Font size is too small.");
46
47     // make sure that freetype has been initialised
48     if (!ft_library) tekThrow(FREETYPE_EXCEPTION, "FreeType
must be initialised before creating a font face.");
49
50     // load the font face from the file
51     if (FT_New_Face(ft_library, filename, face_index, face))
tekThrow(FREETYPE_EXCEPTION, "Failed to create font face.");
52
53     // set the pixel size of the font
54     // setting width to 0 means allow any width, set the height
to the size we wanted
55     if (FT_Set_Pixel_Sizes(*face, 0, face_size))
tekThrow(FREETYPE_EXCEPTION, "Failed to set face pixel size.");
56     return SUCCESS;
57 }
58
59 /**
60 * Get the size of a glyph without loading the bitmap data of
the glyph.
61 * @param face The font face to get the glyph from.
62 * @param glyph The id of the glyph to load, should be the
ascii value of the character you want.

```

```

63  * @param glyph_width The outputted width of the glyph.
64  * @param glyph_height The outputted height of the glyph.
65  * @throws FREETYPE_EXCEPTION if FreeType was not initialised
of the glyph could not be loaded.
66  */
67 exception tekGetGlyphSize(const FT_Face* face, const uint
glyph, uint* glyph_width, uint* glyph_height) {
68     // make sure that the library has been initialised
69     if (!ft_library) tekThrow(FREETYPE_EXCEPTION, "FreeType
must be initialised before loading a glyph.");
70
71     // load the glyph without loading the bitmap data, we only
want the metrics
72     if (FT_Load_Char(*face, glyph,
FT_LOAD_BITMAP_METRICS_ONLY)) tekThrow(FREETYPE_EXCEPTION, "Failed
to get glyph size.");
73
74     // return the metrics
75     if (glyph_width) *glyph_width = (*face)->glyph-
>bitmap.width;
76     if (glyph_height) *glyph_height = (*face)->glyph-
>bitmap.rows;
77     return SUCCESS;
78 }
79
80 /**
81 * Load a glyph temporarily into memory, get the size and
bitmap data of the glyph.
82 * @param face The font face to get the glyph from.
83 * @param glyph_id The id of the glyph, should be the ascii
value of the character you want.
84 * @param glyph The outputted glyph data.
85 * @param glyph_data The outputted bitmap data for the glyph.
86 * @throws FREETYPE_EXCEPTION if FreeType is not initialised or
the glyph couldn't be rendered.
87 */
88 exception tekTempLoadGlyph(const FT_Face* face, const uint
glyph_id, TekGlyph* glyph, byte** glyph_data) {
89     // make sure that the library has been initialised
90     if (!ft_library) tekThrow(FREETYPE_EXCEPTION, "FreeType
must be initialised before loading a glyph.");
91
92     // load the glyph and render the corresponding bitmap

```

```

93     if (FT_Load_Char(*face, glyph_id, FT_LOAD_RENDER))
tekThrow(FREETYPE_EXCEPTION, "Failed to render glyph.");
94
95     // return the data we got
96     if (glyph) {
97         glyph->atlas_x    = 0;
98         glyph->atlas_y    = 0;
99         glyph->width      = (*face)->glyph->bitmap.width;
100        glyph->height     = (*face)->glyph->bitmap.rows;
101        glyph->bearing_x  = (*face)->glyph->bitmap_left;
102        glyph->bearing_y  = (*face)->glyph->bitmap_top;
103        glyph->advance    = (*face)->glyph->advance.x;
104    }
105    if (glyph_data) *glyph_data = (*face)->glyph-
>bitmap.buffer;
106    return SUCCESS;
107 }
108
109 /**
110  * Get the size of the atlas required to contain a font face.
111  * @param face The font face that needs to be contained.
112  * @param atlas_size The outputted size of the atlas as a
square texture, this is the width and height.
113  * @throws FREETYPE_EXCEPTION if atlas size is smaller than the
minimum allowed size.
114 */
115 exception tekGetAtlasSize(const FT_Face* face, uint*
atlas_size) {
116     // init an array to store the width of each character
117     uint char_widths[ATLAS_SIZE];
118
119     // fill array with the width of each character
120     for (uint i = 0; i < ATLAS_SIZE; i++) {
121         uint glyph_width, glyph_height;
122         tekChainThrow(tekGetGlyphSize(face, i, &glyph_width,
&glyph_height));
123         char_widths[i] = glyph_width;
124     }
125
126     // grab some data about the font atlas we want to make
127     const uint char_height = (*face)->size->metrics.height >>
6;
128     uint max_rows = (uint)ceil(sqrt(ATLAS_SIZE));

```

```

129      if (max_rows < ATLAS_MIN_SIZE) tekThrow(FREETYPE_EXCEPTION,
130          "Atlas size is too small.");
130      uint prev_rows = max_rows;
131      uint max_size = max_rows * char_height;
132
133      // this just goes to the next multiple of 4 (OpenGL
134      // textures are aligned to 4 byte intervals)
134      max_size += 3;
135      max_size &= ~3;
136
137      // iterative process to find the best size
138      flag quit = ATLAS_FIRST_TRIAL;
139      while (1) {
140          // initialise some counting variables
141          uint num_rows = 1;
142          uint sum = 0;
143
144          // keep adding widths until we hit the maximum width
145          // (char width * num rows)
145          // when this happens, start a new row
146          // if we go past maximum rows, then we know atlas is
146          // too small
147          for (uint i = 0; i < ATLAS_SIZE; i++) {
148              sum += char_widths[i];
149              if (sum >= max_size) {
150                  num_rows++;
151                  if (num_rows == max_rows) {
152                      quit |= ATLAS_QUIT;
153                      break;
154                  }
155                  sum = char_widths[i];
156              }
157          }
158
159          // keep track of the previous size - this is the size
159          // we want to use if the next
160          prev_rows = max_rows;
161
162          // if we quit early, then the atlas is too small, so
162          // increase the size
163          // however, if we have never quit before, and this
163          // is first time, last attempt is smallest you can get

```

```

164          // if we don't quit early, atlas is too big, so reduce
the size
165          // however, if we have quit before, we have grown
up to this size and finally reached it, so this is the smallest you
can get
166          if ((quit & ATLAS_QUIT) == ATLAS_QUIT) {
167              if ((quit & ATLAS_FIRST_TRIAL) == 0)
168                  if ((quit & ATLAS_QUIT_ALREADY) == 0) break;
169              quit = ATLAS_QUIT_ALREADY;
170              max_rows++;
171          } else {
172              if ((quit & ATLAS_QUIT_ALREADY) ==
ATLAS_QUIT_ALREADY) {
173                  prev_rows--;
174                  max_size = prev_rows * char_height;
175                  max_size += 3;
176                  max_size &= ~3;
177                  break;
178              }
179              quit = 0;
180              max_rows--;
181          }
182
183          // update new maximum size
184          max_size = prev_rows * char_height;
185          max_size += 3;
186          max_size &= ~3;
187      }
188
189      // return atlas size
190      *atlas_size = max_size;
191      return SUCCESS;
192  }
193
194 /**
195  * Create the font atlas and update the glyph data for each
character once it has been added to the font atlas.
196  * @param face The font face that is being used to create the
font atlas.
197  * @param atlas_size The size of the atlas from \ref
tekGetAtlasSize
198  * @param atlas_data A pointer to the buffer containing the
atlas, which will have the glyphs written to it.

```

```

199  * @param glyphs The array of glyphs that will be updated with
the glyph data.
200  * @throws FREETYPE_EXCEPTION if could not load a glyph from
the font face.
201  */
202 exception tekCreateFontAtlasData(const FT_Face* face, const
uint atlas_size, byte** atlas_data, TekGlyph* glyphs) {
203     // grab some initial data
204     const uint char_height = (*face)->size->metrics.height >>
6;
205     uint atlas_x = 0, atlas_y = 0;
206
207     // loop through each of the characters in the atlas
208     for (uint i = 0; i < ATLAS_SIZE; i++) {
209         // load the glyph data and its bitmap
210         byte* glyph_data;
211         tekChainThrow(tekTempLoadGlyph(face, i, &glyphs[i],
&glyph_data));
212
213         // make sure that we aren't gonna overflow the texture
214         if (atlas_x + glyphs[i].width >= atlas_size) {
215             atlas_x = 0;
216             atlas_y += char_height;
217         }
218
219         // set the atlas pointers of the glyph
220         glyphs[i].atlas_x = atlas_x;
221         glyphs[i].atlas_y = atlas_y;
222
223         // copy over the glyph's bitmap into the larger atlas
bitmap
224         for (uint y = 0; y < glyphs[i].height; y++) {
225             for (uint x = 0; x < glyphs[i].width; x++) {
226                 const uint int_index = y * glyphs[i].width + x;
227                 const uint ext_index = (atlas_y + y) *
atlas_size + atlas_x + x;
228                 (*atlas_data)[ext_index] =
glyph_data[int_index];
229             }
230         }
231
232         // adjust the atlas pointer, jumping to next row if and
when needed

```

```

233         atlas_x += glyphs[i].width;
234     }
235
236     return SUCCESS;
237 }
238
239 /**
240  * Create the font atlas texture, this is a texture that
241  * contains every character in a tightly packed area.
242  * Also returns an array of glyphs, which store where on the
243  * atlas each character is.
244  * @param face The font face to create the texture atlas from.
245  * @param texture_id A pointer to a uint which will have the id
246  * of the newly created texture written to it.
247  * @param atlas_size A pointer to a uint which will have the
248  * new atlas size written to it.
249  * @param glyphs A pointer to an array of glyphs which will
250  * have the glyph data written to it.
251  * @throws MEMORY_EXCEPTION if malloc() fails.
252  * @throws FREETYPE_EXCEPTION if failed to load font.
253  */
254 exception tekCreateFontAtlasTexture(const FT_Face* face, uint*
255 texture_id, uint* atlas_size, TekGlyph* glyphs) {
256     tekChainThrow(tekGetAtlasSize(face, atlas_size));
257
258     // allocate enough data for the atlas texture
259     byte* atlas_data = (byte*)malloc(*atlas_size *
260     *atlas_size);
261     if (!atlas_data) tekThrow(MEMORY_EXCEPTION, "Failed to
262     allocate memory for atlas data.");
263
264     // pass empty buffer and fill it with atlas texture data
265     const exception result = tekCreateFontAtlasData(face,
266     *atlas_size, &atlas_data, glyphs);
267     if (result) {
268         free(atlas_data);
269         tekChainThrow(result);
270     }
271
272     // create a new opengl texture for the atlas
273     glGenTextures(1, texture_id);
274     glBindTexture(GL_TEXTURE_2D, *texture_id);
275
276

```

```

267     // set the properties of the texture
268     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
269     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
270     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
271     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
272
273     // write atlas texture data to buffer
274     glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, (int)*atlas_size,
(int)*atlas_size, 0, GL_RED, GL_UNSIGNED_BYTE, atlas_data);
275
276     // free the atlas data as it is no longer needed
277     free(atlas_data);
278     return SUCCESS;
279 }
280
281 /**
282 * Create a new bitmap font from a true type font file. The
files contain multiple faces such as italic, bold.
283 * So you need to also specify which font you want to use with
a font index.
284 * @param filename The name of the .ttf file to be loaded.
285 * @param face_index The index of the face to use (use 0 if
unsure)
286 * @param face_size The size of the face in pixels.
287 * @param bitmap_font A pointer to a TekBitmapFont struct that
will have the font data written to it.
288 * @throws FREETYPE_EXCEPTION if failed to load the font.
289 */
290 exception tekCreateBitmapFont(const char* filename, const uint
face_index, const uint face_size, TekBitmapFont* bitmap_font) {
291     // load the font face
292     FT_Face face;
293     tekChainThrow(tekCreateFontFace(filename, face_index,
face_size, &face));
294
295     // create the font atlas texture
296     tekChainThrow(tekCreateFontAtlasTexture(&face,
&bitmap_font->atlas_id, &bitmap_font->atlas_size, &bitmap_font-
>glyphs));

```

```

297
298     // store original size
299     bitmap_font->original_size = face->size->metrics.height >>
6;
300     tekDeleteFontFace(face); // delete font face, we only need
the atlas from it.
301     return SUCCESS;
302 }

```

tekgl/font.h

```

1  #pragma once
2
3  #include "../core/exception.h"
4  #include "../tekgl.h"
5
6  #include <freetype/freetype.h>
7  #define FT_FREETYPE_H
8
9  #define ATLAS_SIZE      256
10 #define MIN_FONT_SIZE 3
11
12 /**
13  * Delete a font face, wrapper around normal method to keep
naming convention.
14 */
15 #define tekDeleteFontFace(face) FT_Done_Face(face)
16
17 typedef struct TekGlyph {
18     uint atlas_x, atlas_y;
19     unsigned short width, height;
20     unsigned short bearing_x, bearing_y;
21     unsigned short advance;
22 } TekGlyph;
23
24 typedef struct TekBitmapFont {
25     uint atlas_id;
26     uint atlas_size;
27     uint original_size;
28     TekGlyph glyphs[ATLAS_SIZE];
29 } TekBitmapFont;
30
31 exception tekCreateFreeType();
32 void tekDeleteFreeType();

```

```

33 exception tekCreateFontFace(const char* filename, uint
face_index, uint face_size, FT_Face* face);
34 exception tekTempLoadGlyph(const FT_Face* face, uint glyph_id,
TekGlyph* glyph, byte** glyph_data);
35 //exception tekCreateFontAtlasTexture(const FT_Face* face, uint*
texture_id, TekGlyph* glyphs);
36 exception tekCreateBitmapFont(const char* filename, uint
face_index, uint face_size, TekBitmapFont* bitmap_font);

```

tekgl/manager.c

```

1 #include "manager.h"
2
3 #include <stdlib.h>
4
5 #include "entity.h"
6 #include "text.h"
7 #include "../tekgui/primitives.h"
8 #include "glad/glad.h"
9 #include "GLFW/glfw3.h"
10 #include "../core/list.h"
11
12 static GLFWwindow* tek_window = 0;
13 static int tek_window_width = 0;
14 static int tek_window_height = 0;
15 static vec3 tek_window_colour = { 0.0f, 0.0f, 0.0f };
16
17 static List tek_fb_funcs = {0, 0};
18 static List tek_delete_funcs = {0, 0};
19 static List tek_gl_load_funcs = {0, 0};
20 static List tek_key_funcs = {0, 0};
21 static List tek_char_funcs = {0, 0};
22 static List tek_mmove_funcs = {0, 0};
23 static List tek_mbutton_funcs = {0, 0};
24 static List tek_mscroll_funcs = {0, 0};
25 static GLFWcursor* crosshair_cursor;
26
27 /**
28 * Add a framebuffer callback which will be called whenever the
framebuffer/window is resized.
29 * @param callback The callback function.
30 * @throws LIST_EXCEPTION if could not add function to list.
31 */

```

```

32 exception tekAddFramebufferCallback(const
TekFramebufferCallback callback) {
33     if (!tek_fb_funcs.length) { // create list if not yet
created
34         listCreate(&tek_fb_funcs);
35     }
36
37     tekChainThrow(listAddItem(&tek_fb_funcs, callback)); // add
callback to list
38     return SUCCESS;
39 }
40
41 /**
42 * Add a framebuffer callback which is called every time the
window is resized.
43 * This gives the new size of the portion of the window which
is being rendered, and doesn't include borders.
44 * @param window The GLFW window which is being resized.
45 * @param width The new width of the framebuffer.
46 * @param height The new height of the framebuffer.
47 */
48 void tekManagerFramebufferCallback(GLFWwindow* window, const
int width, const int height) {
49     if (window != tek_window) return;
50
51     // update these so we can retrieve at any time.
52     tek_window_width = width;
53     tek_window_height = height;
54
55     // update the size of the viewport
56     glViewport(0, 0, width, height);
57
58     // call any user-created framebuffer callbacks.
59     const ListItem* item = 0;
60     foreach(item, (&tek_fb_funcs), {
61         const TekFramebufferCallback callback =
(TekFramebufferCallback)item->data;
62         callback(width, height);
63     });
64 }
65
66 /**

```

```

67  * Add a delete function which is called when the program is
terminating.
68  * @param delete_func The function to call on termination.
69  * @throws LIST_EXCEPTION if could not add function to list.
70  */
71 exception tekAddDeleteFunc(const TekDeleteFunc delete_func) {
72     if (!tek_delete_funcs.length)
73         listCreate(&tek_delete_funcs);
74
75     // add delete func to a list that we can iterate over on
cleanup
76     tekChainThrow(listAddItem(&tek_delete_funcs, delete_func));
77     return SUCCESS;
78 }
79
80 /**
81 * Add a GL load function, this is called after glfw + glad
have been initialised.
82 * @param gl_load_func The function to be called during
loading.
83 * @throws LIST_EXCEPTION if could not add function to list.
84 */
85 exception tekAddGLLoadFunc(const TekGLLoadFunc gl_load_func) {
86     if (!tek_gl_load_funcs.length)
87         listCreate(&tek_gl_load_funcs);
88
89     // add gl load func to a list that we can iterate over on
cleanup
90     tekChainThrow(listAddItem(&tek_gl_load_funcs,
gl_load_func));
91     return SUCCESS;
92 }
93
94 /**
95 * The main callback for all key press events, goes on to call
any user-created key callbacks.
96 * @param window The window where the key press is occurring.
97 * @param key The key being pressed as a GLFW_KEY_...
enum/macro.
98 * @param scancode The scancode of the key
99 * @param action The action e.g. press, release, repeat.
100 * @param mods Any modifiers acting on the key.
101 */

```

```

102 void tekManagerKeyCallback(GLFWwindow* window, const int key,
103   const int scancode, const int action, const int mods) {
104     if (window != tek_window) return;
105     // for each key callback.
106     const ListItem* item = 0;
107     foreach(item, (&tek_key_funcs), {
108       const TekKeyCallback callback = (TekKeyCallback)item-
109         >data; // get the callback.
110       callback(key, scancode, action, mods); // call it.
111     });
112
113 /**
114  * The main callback for char events, e.g. listen for typing
115  * with shift/caps lock affecting keys.
116  * Goes on to call all user created char callbacks.
117  * @param window The GLFW window where the typing is happening.
118  * @param codepoint The character being pressed as ascii code.
119 */
120 static void tekManagerCharCallback(GLFWwindow* window, uint
121 codepoint) {
122   if (window != tek_window) return;
123   // for each callback in list
124   const ListItem* item = 0;
125   foreach(item, (&tek_char_funcs), {
126     const TekCharCallback callback = (TekCharCallback)item-
127       >data; // get callback
128     callback(codepoint); // call it
129   });
130 }
131 /**
132  * The main callback for mouse movement events that will go on
133  * to trigger all user-created callbacks of this type.
134  * @param window The GLFW window where the event happened.
135  * @param x The x position of the mouse after moving.
136  * @param y The y position of the mouse after moving.
137 */
138 static void tekManagerMouseMoveCallback(GLFWwindow* window,
139   const double x, const double y) {
140   if (window != tek_window) return;

```

```

138
139     // for each item
140     const ListItem* item = 0;
141     foreach(item, (&tek_mmove_funcs), {
142         const TekMousePosCallback callback =
(TekMousePosCallback)item->data; // get the callback
143         callback(x, y); // call it
144     });
145 }
146
147 /**
148 * The main callback for mouse buttons that will go on to
trigger user-created callbacks.
149 * @param window The GLFW window pointer that is being clicked.
150 * @param button The button being clicked.
151 * @param action The type of click e.g. press or release.
152 * @param mods The modifiers acting on the button.
153 */
154 static void tekManagerMouseButtonCallback(GLFWwindow* window,
const int button, const int action, const int mods) {
155     if (window != tek_window) return;
156
157     // for each item
158     const ListItem* item = 0;
159     foreach(item, (&tek_mbutton_funcs), {
160         const TekMouseButtonCallback callback =
(TekMouseButtonCallback)item->data; // get the callback
161         callback(button, action, mods); // call it
162     });
163 }
164
165 /**
166 * The root callback for mouse scroll events, which triggers
all user-created events.
167 * @param window The GLFW window pointer.
168 * @param x_offset The amount of scroll in the x direction.
169 * @param y_offset The amount of scroll in the y direction.
170 */
171 static void tekManagerMouseScrollCallback(GLFWwindow* window,
const double x_offset, const double y_offset) {
172     if (window != tek_window) return; // only respond to events
from this window i guess
173

```

```

174     // list for loop
175     const ListItem* item = 0;
176     foreach(item, (&tek_mscroll_funcs), {
177         const TekMouseScrollCallback callback =
178             (TekMouseScrollCallback)item->data; // get callback
179         callback(x_offset, y_offset); // call it
180     });
181
182 /**
183 * Add a callback for every time a key is pressed on the
184 * keyboard.
185 * @param callback The callback function.
186 * @throws LIST_EXCEPTION if the function could not be added to
187 * the list.
188 */
189 exception tekAddKeyCallback(const TekKeyCallback callback) {
190     if (!tek_key_funcs.length) // if length == 0, then create
191         the list as it has not yet been
192         listCreate(&tek_key_funcs);
193
194
195 /**
196 * Add a 'char' callback, which is called whenever the user
197 * types. It is affected by shift/caps lock etc.
198 * @param callback The callback function.
199 * @throws LIST_EXCEPTION if the function could not be added to
200 * the list of callbacks.
201 */
202 exception tekAddCharCallback(const TekCharCallback callback) {
203     if (!tek_char_funcs.length) // if length == 0, then list is
204         yet to be created.
205         listCreate(&tek_char_funcs);
206
207
208 /**

```

```

209  * Add a callback that is triggered every time the mouse is
210 * moved.
211 * @param callback The callback function.
212 * @throws LIST_EXCEPTION if the callback could not be added to
213 * the list of callbacks.
214 */
215 exception tekAddMousePosCallback(const TekMousePosCallback
216 callback) {
217     if (!tek_mmove_funcs.length) // if length == 0, then list
218 is yet to be created.
219     listCreate(&tek_mmove_funcs);
220
221 /**
222 * Add a mouse button callback that is triggered every time the
223 * mouse is clicked.
224 * @param callback The callback function
225 * @throws LIST_EXCEPTION if could not add callback to list.
226 */
227 exception tekAddMouseButtonCallback(const
228 TekMouseButtonCallback callback) {
229     if (!tek_mbutton_funcs.length) // if length == 0, then list
229 has not yet been created.
230     listCreate(&tek_mbutton_funcs);
231
232 /**
233 * Add a callback that will be triggered every time the mouse
234 * is scrolled.
235 * @param callback The function to call.
236 * @throws LIST_EXCEPTION if could not add callback to list of
237 * callbacks.
238 */
239 exception tekAddMouseScrollCallback(const
240 TekMouseScrollCallback callback) {
241     if (!tek_mscroll_funcs.length) // if length == 0, then
uninitialised.

```

```

241         listCreate(&tek_mscroll_funcs);
242
243         tekChainThrow(listAddItem(&tek_mscroll_funcs, callback));
244         return SUCCESS;
245     }
246
247 /**
248 * Set the mode of the cursor. Should be one of two modes:
249 * DEFAULT_CURSOR or CROSSHAIR_CURSOR, assumes default cursor
250 if invalid cursor mode specified.
251 * @param cursor_mode The cursor mode to use.
252 */
253 void tekSetCursor(const flag cursor_mode) {
254     switch (cursor_mode) {
255         case CROSSHAIR_CURSOR: // if crosshair cursor, set the
256         cursor to crosshair cursor
257             glfwSetCursor(tek_window, crosshair_cursor);
258             break;
259         default: // NULL = default cursor.
260             glfwSetCursor(tek_window, NULL);
261     }
262 }
263 /**
264 * Set the colour of the window background when nothing is
265 drawn.
266 * @param colour The colour to use.
267 */
268 void tekSetWindowColour(vec3 colour) {
269     glm_vec3_copy(colour, tek_window_colour); // copy into
270 window colour variable.
271 }
272 /**
273 * Set the draw mode of the window, should be one of:
274 * DRAW_MODE_NORMAL - draw everything as expected.
275 * DRAW_MODE_GUI - draw everything as gui.
276 * This changes how depth is rendered, gui elements are all
277 rendered at the same level.
278 * Typical depth test will not work, so setting gui mode will
279 force new things to appear above old things.
280 * @param draw_mode
281 */

```

```

278 void tekSetDrawMode(const flag draw_mode) {
279     if (draw_mode == DRAW_MODE_NORMAL) {
280         glDepthFunc(GL_LESS); // obscure things which are
behind
281         return;
282     }
283
284     if (draw_mode == DRAW_MODE_GUI) {
285         glDepthFunc(GL_ALWAYS); // always draw over things
286     }
287 }
288
289 /**
290 * Initialise the TekPhysics window.
291 * @param window_name The title to be displayed for the window.
292 * @param window_width The initial width of the window.
293 * @param window_height The initial height of the window.
294 * @throws GLFW_EXCEPTION if GLFW failed to initialise or
create the window.
295 * @throws GLAD_EXCEPTION if GLAD failed to load.
296 */
297 exception tekInit(const char* window_name, const int
window_width, const int window_height) {
298     // initialise glfw
299     if (!glfwInit()) tekThrow(GLFW_EXCEPTION, "GLFW failed to
initialise.");
300
301     // create a window with opengl and store some info about it
302     tek_window = glfwCreateWindow(window_width, window_height,
window_name, NULL, NULL);
303     if (!tek_window) {
304         glfwTerminate();
305         tekThrow(GLFW_EXCEPTION, "GLFW failed to create
window.");
306     }
307
308     // make this window as current context, so it will receive
opengl draw calls
309     glfwMakeContextCurrent(tek_window);
310
311     // load glad, allows us to use opengl functions
312     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
tekThrow(GLAD_EXCEPTION, "GLAD failed to load loader.");

```

```

313
314     // update framebuffer size and callbacks
315     glfwGetFramebufferSize(tek_window, &tek_window_width,
316                             &tek_window_height);
316     glfwSetFramebufferSizeCallback(tek_window,
317                                   tekManagerFramebufferCallback);
317     glfwSetKeyCallback(tek_window, tekManagerKeyCallback);
318     glfwSetCharCallback(tek_window, tekManagerCharCallback);
319     glfwSetCursorPosCallback(tek_window,
320                             tekManagerMouseMoveCallback);
320     glfwSetMouseButtonCallback(tek_window,
321                               tekManagerMouseButtonCallback);
321     glfwSetScrollCallback(tek_window,
322                           tekManagerMouseScrollCallback);
322
323     crosshair_cursor =
324     glfwCreateStandardCursor(GLFW_RESIZE_ALL_CURSOR);
324
325     tekChainThrow(tekCreateFreeType());
326
327     // run all the callbacks for when opengl loaded.
328     const ListItem* item;
329     foreach(item, (&tek_gl_load_funcs), {
330         const TekGLLoadFunc gl_load_func = (TekGLLoadFunc)item-
331 >data;
331         tekChainThrow(gl_load_func());
332     });
333
334     tekManagerFramebufferCallback(tek_window, window_width,
335                                 window_height);
335
336     // dont render stuff that is behind other objects.
337     glEnable(GL_DEPTH_TEST);
338     glDepthFunc(GL_LESS);
339
340     // dont render stuff that is facing the wrong way
341     glEnable(GL_CULL_FACE);
342
343     // blend translucent colours nicely.
344     glEnable(GL_BLEND);
345     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
346
347     return SUCCESS;

```

```

348 }
349 /**
350 * Return whether the window should remain open.
351 * @return 1 if the window is still open, 0 if the X button has
352 been pressed.
353 */
354 flag tekRunning() {
355     // return whether tekgl should continue running
356     return glfwWindowShouldClose(tek_window) ? 0 : 1;
357 }
358
359 /**
360 * Swap buffers and poll events. Should be called every frame / every
361 loop of the main loop.
362 * @throws NOTHING not sure why it's not a void function.
363 */
364 exception tekUpdate() {
365     // swap buffers - clears front buffer and displays back
366 buffer to screen
367     glfwSwapBuffers(tek_window);
368
369     // push any events that have occurred - key presses, close
370 window etc.
371     glfwPollEvents();
372
373     // clear framebuffer for next draw call
374     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
375     glClearColor(tek_window_colour[0], tek_window_colour[1],
376 tek_window_colour[2], 1.0f);
377
378     return SUCCESS;
379 }
380
381 /**
382 * Free memory allocated by supporting structures for the
383 manager code, for example different callback function lists.
384 */
385 void tekDelete() {
386     // run all cleanup functions
387     const ListItem* item = 0;
388     foreach(item, (&tek_delete_funcs), {

```

```

384         const TekDeleteFunc delete_func = (TekDeleteFunc)item-
>data;
385         delete_func();
386     });
387
388     // clean up memory allocated by callback functions
389     listDelete(&tek_fb_funcs);
390     listDelete(&tek_delete_funcs);
391     listDelete(&tek_gl_load_funcs);
392     listDelete(&tek_key_funcs);
393     listDelete(&tek_mmove_funcs);
394     listDelete(&tek_mbutton_funcs);
395     listDelete(&tek_msroll_funcs);
396
397     tekDeleteFreeType();
398
399     glfwDestroyCursor(crosshair_cursor);
400
401     // destroy the glfw window and context
402     glfwDestroyWindow(tek_window);
403     glfwTerminate();
404 }
405
406 /**
407 * Get the current size of the window.
408 * @param window_width The outputted width.
409 * @param window_height The outputted height.
410 */
411 void tekGetWindowSize(int* window_width, int* window_height) {
412     // pointers :D
413     *window_width = tek_window_width;
414     *window_height = tek_window_height;
415 }
416
417 /**
418 * Set the mouse mode of the window. Can be one of two types:
419 * MOUSE_MODE_CAMERA - make the mouse disappear and unable to
leave the screen.
420 * MOUSE_MODE_NORMAL - make the mouse behave as expected.
421 * If neither one is used, then normal mode is assumed.
422 * @param mouse_mode The mouse mode as specified above
423 */
424 void tekSetMouseMode(const flag mouse_mode) {

```

```

425     // cannot change mouse mode unless window is focused
426     glfwFocusWindow(tek_window);
427     switch (mouse_mode) {
428         case MOUSE_MODE_CAMERA:
429             if (glfwRawMouseMotionSupported())
430                 glfwSetInputMode(tek_window, GLFW_RAW_MOUSE_MOTION,
GLFW_TRUE); // raw motion
431             glfwSetInputMode(tek_window, GLFW_CURSOR,
GLFW_CURSOR_DISABLED); // hide cursor
432             break;
433         case MOUSE_MODE_NORMAL:
434         default:
435             glfwSetInputMode(tek_window, GLFW_RAW_MOUSE_MOTION,
GLFW_FALSE); // normal mouse movement
436             glfwSetInputMode(tek_window, GLFW_CURSOR,
GLFW_CURSOR_NORMAL); // show cursor
437     }
438 }
```

tekgl/manager.h

```

1 #pragma once
2
3 #include <cglm/ivec3.h>
4
5 #include "../tekgl.h"
6 #include "../core/exception.h"
7
8 #define tek_init __attribute__((constructor)) static void
9
10#define MOUSE_MODE_NORMAL 0
11#define MOUSE_MODE_CAMERA 1
12
13#define DEFAULT_CURSOR 0
14#define CROSSHAIR_CURSOR 1
15
16#define DRAW_MODE_NORMAL 0
17#define DRAW_MODE_GUI 1
18
19typedef void(*TekFramebufferCallback)(int framebuffer_width, int
framebuffer_height);
20typedef void(*TekDeleteFunc)(void);
21typedef exception(*TekGLLoadFunc)(void);
```

```

22 typedef void(*TekKeyCallback)(int key, int scancode, int action,
int mods);
23 typedef void(*TekCharCallback)(uint codepoint);
24 typedef void(*TekMousePosCallback)(double x, double y);
25 typedef void(*TekMouseButtonCallback)(int button, int action,
int mods);
26 typedef void(*TekMouseScrollCallback)(double x_offset, double
y_offset);
27
28 exception tekInit(const char* window_name, int window_width, int
window_height);
29 flag tekRunning();
30 exception tekUpdate();
31 void tekDelete();
32
33 void tekGetWindowSize(int* window_width, int* window_height);
34
35 void tekSetMouseMode(flag mouse_mode);
36 void tekSetCursor(flag cursor_mode);
37 void tekSetWindowColour(vec3 colour);
38 void tekSetDrawMode(flag draw_mode);
39
40 exception tekAddFramebufferCallback(TekFramebufferCallback
callback);
41 exception tekAddKeyCallback(TekKeyCallback callback);
42 exception tekAddCharCallback(TekCharCallback callback);
43 exception tekAddDeleteFunc(TekDeleteFunc delete_func);
44 exception tekAddGLLoadFunc(TekGLLoadFunc gl_load_func);
45 exception tekAddMousePosCallback(TekMousePosCallback callback);
46 exception tekAddMouseButtonCallback(TekMouseButtonCallback
callback);
47 exception tekAddMouseScrollCallback(TekMouseScrollCallback
callback);

```

tekgl/material.c

```

1 #include "material.h"
2
3 #include <stdio.h>
4 #include <string.h>
5
6 #include "shader.h"
7 #include "../core/yml.h"
8 #include <cglm/vec2.h>

```

```

9  #include <cglm/vec3.h>
10 #include <cglm/vec4.h>
11
12 #include "texture.h"
13
14 #define UINTEGER_DATA 0
15 #define UFLOAT_DATA 1
16 #define TEXTURE_DATA 2
17 #define VEC2_DATA 3
18 #define VEC3_DATA 4
19 #define VEC4_DATA 5
20 #define UNKNOWN_DATA 6
21 #define UYML_DATA 7
22 #define WILDCARD_DATA 8
23
24 #define RGBA_DATA 0
25 #define XYZW_DATA 1
26 #define UV_DATA 2
27 #define RGBA_WORD_DATA 3
28
29 #define MODEL_MATRIX_WILDCARD "$tek_model_matrix"
30 #define VIEW_MATRIX_WILDCARD "$tek_view_matrix"
31 #define PROJECTION_MATRIX_WILDCARD "$tek_projection_matrix"
32 #define CAMERA_POSITION_WILDCARD "$tek_camera_position"
33
34 /**
35  * Create a uniform for vector data that is made of multiple
36  * yml key value pairs.
37  * @param hashtable The yml data containing the vector.
38  * @param num_items The number of items in the vector.
39  * @param keys_order An array of strings that states the order
40  * to access the keys of the vector in.
41  * @param uniform A pointer to the uniform to update with the
42  * vector data.
43  * @throws OPENGL_EXCEPTION if the number of items is not in
44  * the range [2,4], as this is not a valid cglm vector any more.
45  */
46 exception tekCreateVecUniform(const HashTable* hashtable, const
47 uint num_items, const char** keys_order, TekMaterialUniform*
48 uniform) {
49     // enforce minmax length

```

```

44      if ((num_items < 2) || (num_items > 4))
tekThrow(OPENGL_EXCEPTION, "Cannot create vector with more than 4 or
less than 2 items.");
45      exception tek_exception = SUCCESS;
46
47      // loop thru keys and access value of each key
48      float vector[4]; // maximum size is 4, no point m allocateing
to save only 2 floats of space for this temporary buffer
49      for (uint i = 0; i < num_items; i++) {
50          // construct the vector one piece at a time
51          YmlData* data;
52          tek_exception = hashtableGet(hashtable, keys_order[i],
&data); tekChainBreak(tek_exception);
53          double number;
54          tek_exception = ymlDataToFloat(data, &number);
tekChainBreak(tek_exception);
55          vector[i] = (float)number;
56      }
57      tekChainThrow(tek_exception);
58
59      // NOW m allocate because this cant be stack memory anyway
60      float* uniform_data = malloc(num_items * sizeof(float));
61      uniform->data = uniform_data;
62      if (!uniform->data) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for vector.");
63
64      // copy the temp buffer over.
65      for (uint i = 0; i < num_items; i++) {
66          memcpy(uniform_data + i, vector + i, sizeof(float));
67      }
68
69      // macro is ordered, so VEC3_DATA is one more than
VEC2_DATA etc.
70      uniform->type = VEC2_DATA + (num_items - 2);
71      return SUCCESS;
72  }
73
74 /**
75  * Create a single uniform to add to the uniform array in the
material.
76  * @param uniform_name The name of the uniform.
77  * @param data_type The type of data stored in the uniform.
78  * @param data The actual data being stored in the uniform.

```



```

111         break;
112     // texture data needs to have a texture created with
113     opengl
114     case TEXTURE_DATA:
115         const char* filepath = data;
116         uint* texture_id = (uint*)malloc(sizeof(uint)); // /
117         // allocate space for new uint
118         if (!texture_id) {
119             tek_exception = MEMORY_EXCEPTION;
120             tekExcept(tek_exception, "Failed to allocate
121             memory for texture id.");
122             break;
123         }
124         tek_exception = tekCreateTexture(filepath,
125         texture_id); // create the texture from provided data
126         if (tek_exception) {
127             free(texture_id);
128             tekChainBreak(tek_exception);
129             break;
130         }
131         (*uniform)->data = texture_id;
132         break;
133     // wildcard such as $tek_projection_matrix etc.
134     case WILDCARD_DATA:
135         // no data stored here, just tells us to pass in
136         camera data to the shader and whatnot
137         (*uniform)->data = 0;
138         const char* wildcard = data;
139         if (!strcmp(wildcard, MODEL_MATRIX_WILDCARD))
140             (*uniform)->type = MODEL_MATRIX_DATA;
141         else if (!strcmp(wildcard, VIEW_MATRIX_WILDCARD))
142             (*uniform)->type = VIEW_MATRIX_DATA;
143         else if (!strcmp(wildcard,
144             PROJECTION_MATRIX_WILDCARD))
145             (*uniform)->type = PROJECTION_MATRIX_DATA;
146         else if (!strcmp(wildcard,
147             CAMERA_POSITION_WILDCARD))
148             (*uniform)->type = CAMERA_POSITION_DATA;
149         else {
150             tek_exception = YML_EXCEPTION;
151             tekExcept(tek_exception, "Unrecognised wilcard
152             used.");
153             break;
154         }
155         break;

```

```

146           // multiple data points such as colours, positions etc
stored here.
147       case UYML_DATA:
148           const HashTable* hashtable = data;
149           char** keys = 0;
150           tekChainBreak(hashtableGetKeys(hashtable, &keys));
151           const uint num_items = hashtable->num_items;
152
153           // find a key that reveals what kind of vector we
are working with
154           // yml is unordered so need to linear search it
155           flag vec_mode = UNKNOWN_DATA;
156           for (uint i = 0; i < num_items; i++) {
157               if (!strcmp("r", keys[i])) { // checking for
rgb(a) mode
158                   vec_mode = RGBA_DATA;
159                   break;
160               }
161               if (!strcmp("x", keys[i])) { // checking for
xyz(w) mode
162                   vec_mode = XYZW_DATA;
163                   break;
164               }
165               if (!strcmp("u", keys[i])) { // checking for uv
mode
166                   vec_mode = UV_DATA;
167                   break;
168               }
169               if (!strcmp("red", keys[i])) { // checking for
literal red green blue (alpha) mode
170                   vec_mode = RGBA_WORD_DATA;
171                   break;
172               }
173           }
174
175           // if no mode is found, throw an error
176           if (vec_mode == UNKNOWN_DATA) {
177               tek_exception = YML_EXCEPTION;
178               tekExcept(tek_exception, "Bad layout for vector
data.");
179               free(keys);
180               break;
181           }

```

```

182
183     // enforce some error checking stuff
184     if ((vec_mode == RGBA_DATA) && (num_items < 3)) {
185         tek_exception = YML_EXCEPTION;
186         tekExcept(tek_exception, "Incorrect number of
187 items for RGBA expression.");
188     }
189     if ((vec_mode == XYZW_DATA) && (num_items < 3)) {
190         tek_exception = YML_EXCEPTION;
191         tekExcept(tek_exception, "Incorrect number of
192 items for XYZW expression.");
193     }
194     if ((vec_mode == UV_DATA) && (num_items < 2)) {
195         tek_exception = YML_EXCEPTION;
196         tekExcept(tek_exception, "Incorrect number of
197 items for UV expression.");
198     }
199     if ((vec_mode == RGBA_WORD_DATA) && (num_items <
200 3)) {
201         tek_exception = YML_EXCEPTION;
202         tekExcept(tek_exception, "Incorrect number of
203 items for red green blue alpha expression.");
204     }
205     if (tek_exception) {
206         free(keys);
207         break;
208     }
209
210     // hard code the order to access the keys in
211     // as previously stated, the keys are unordered so
212     // cant just do data[0] data[1] etc.
213     char* keys_order[4];
214     if (vec_mode == RGBA_DATA) {
215         keys_order[0] = "r";
216         keys_order[1] = "g";
217         keys_order[2] = "b";
218         keys_order[3] = "a";
219     }
220     if (vec_mode == XYZW_DATA) {
221         keys_order[0] = "x";
222         keys_order[1] = "y";
223         keys_order[2] = "z";
224         keys_order[3] = "w";

```

```

219         }
220         if (vec_mode == UV_DATA) {
221             keys_order[0] = "u";
222             keys_order[1] = "v";
223         }
224         if (vec_mode == RGBA_WORD_DATA) {
225             keys_order[0] = "red";
226             keys_order[1] = "green";
227             keys_order[2] = "blue";
228             keys_order[3] = "alpha";
229         }
230
231         // pass onto subroutine cuz it would be too long
232         // although this function is already stupid long
233         tek_exception = tekCreateVecUniform(hashtable,
234                                         num_items, keys_order, *uniform);
235         free(keys);
236         tekChainBreak(tek_exception);
237         break;
238     }
239     if (tek_exception) {
240         free(uniform_name_copy);
241         free(*uniform);
242     }
243     return tek_exception;
244 }
245 /**
246 * Delete a material uniform from memory.
247 * @param uniform The uniform to delete.
248 */
249 void tekDeleteUniform(TekMaterialUniform* uniform) {
250     if (uniform->name) free(uniform->name); // avoid freeing
null pointers cuz it usually has bad consequences
251     if (uniform->data) free(uniform->data);
252     free(uniform);
253 }
254
255 /**
256 * Create a material from a file, a collection of shaders and
uniforms to texture a mesh.
257 * @param filename The filename of the YAML file containing the
material data.

```

```

258     * @param material A pointer to a TekMaterial struct which will
259     * be overwritten with the material data.
260     * @throws FILE_EXCEPTION if file could not be read.
261     * @throws MEMORY_EXCEPTION if malloc() fails.
262     */
263 exception tekCreateMaterial(const char* filename, TekMaterial*
material) {
264     // load up the material yml file
265     YmlFile material_yml = {};
266     ymlCreate(&material_yml);
267     tekChainThrow(ymlReadFile(filename, &material_yml));
268
269     // read some stuff that has to exist
270     YmlData* vs_data = 0;
271     YmlData* fs_data = 0;
272     YmlData* gs_data = 0;
273     tekChainThrow(ymlGet(&material_yml, &vs_data, "shaders",
274                         "vertex_shader"));
275     tekChainThrow(ymlGet(&material_yml, &fs_data, "shaders",
276                         "fragment_shader"));
277
278     // geometry shader is optional, check if we have one, don't
279     // throw error otherwise.
280     exception check_for_geometry = ymlGet(&material_yml,
281                                             &gs_data, "shaders", "geometry_shader");
282     flag has_geometry = 0;
283     if (check_for_geometry == SUCCESS)
284         has_geometry = 1;
285
286     char* vertex_shader;
287     char* fragment_shader;
288
289     exception tek_exception = SUCCESS;
290
291     // read vertex and fragment shaders
292     tek_exception = ymlDataToString(vs_data, &vertex_shader);
293     if (tek_exception) {
294         free(vertex_shader);
295         ymlDelete(&material_yml);
296         tekChainThrow(tek_exception);
297     }
298
299     tek_exception = ymlDataToString(fs_data, &fragment_shader);

```

```

295     if (tek_exception) {
296         free(vertex_shader);
297         free(fragment_shader);
298         ymlDelete(&material_yml);
299         tekChainThrow(tek_exception);
300     }
301
302     uint shader_program_id = 0;
303
304     // read geometry shader if exists
305     // if yes, also need to call a different shader program
function to use it
306     if (has_geometry) {
307         char* geometry_shader;
308         tekChainThrowThen(ymlDataToString(gs_data,
&geometry_shader), {
309             free(vertex_shader);
310             free(fragment_shader);
311             free(geometry_shader);
312             ymlDelete(&material_yml);
313         });
314         tek_exception =
tekCreateShaderProgramVGF(vertex_shader, geometry_shader,
fragment_shader, &shader_program_id);
315         free(geometry_shader);
316     } else {
317         tek_exception = tekCreateShaderProgramVF(vertex_shader,
fragment_shader, &shader_program_id);
318     }
319
320     // no longer needed once the shader is compiled
321     free(vertex_shader);
322     free(fragment_shader);
323     if (tek_exception) {
324         ymlDelete(&material_yml);
325         tekDeleteShaderProgram(shader_program_id);
326         tekChainThrow(tek_exception);
327     }
328
329     // set the shader program id
330     material->shader_program_id = shader_program_id;
331

```

```

332      // according to Comment Buggerer v1.1, i am 39.932%
finished writing ts (352/586 functions still remain)
333      // this is why u should always write comments as you go,
not all at the end
334      // alas i still ignore that
335      char** keys = 0;
336      uint num_keys;
337      tek_exception = ymlGetKeys(&material_yml, &keys, &num_keys,
"uniforms");
338      if (tek_exception) {
339          ymlDelete(&material_yml);
340          tekDeleteShaderProgram(shader_program_id);
341          tekChainThrow(tek_exception);
342      }
343
344      // for however many keys there are, allocate that much
space
345      material->num_uniforms = num_keys;
346      material->uniforms = (TekMaterialUniform**)calloc(num_keys,
sizeof(TekMaterialUniform*));
347      if (!material->uniforms) {
348          ymlDelete(&material_yml);
349          tekDeleteShaderProgram(shader_program_id);
350          tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for uniforms.");
351      }
352
353      // for each key, load in the corresponding uniform
354      for (uint i = 0; i < num_keys; i++) {
355          YmlData* uniform_yml;
356          // archaic exception handling!!! (blast from the past)
357          tek_exception = ymlGet(&material_yml, &uniform_yml,
"uniforms", keys[i]); tekChainBreak(tek_exception);
358          flag uniform_type;
359          switch (uniform_yml->type) { // act different according
to data type. (is this polymorphism?)
360              case YMIL_DATA:
361                  uniform_type = UYML_DATA;
362                  break;
363              case STRING_DATA:
364                  const char* string = uniform_yml->value;
365                  if (string[0] == '$') {
366                      uniform_type = WILDCARD_DATA;

```

```

367         } else {
368             uniform_type = TEXTURE_DATA;
369         }
370         break;
371     case INTEGER_DATA:
372         uniform_type = UINTEGER_DATA;
373         break;
374     case FLOAT_DATA:
375         uniform_type = UFLOAT_DATA;
376         break;
377     }
378
379     // create uniform
380     TekMaterialUniform* uniform;
381     tek_exception = tekCreateUniform(keys[i], uniform_type,
382     uniform_yml->value, &uniform);
383     tekChainBreak(tek_exception);
384
385     // write uniform at that index.
386     material->uniforms[i] = uniform;
387 }
388
389 // if something went wrong, clean up after ourselves
390 if (tek_exception) {
391     for (uint i = 0; i < num_keys; i++) {
392         if (!material->uniforms[i]) break;
393         tekDeleteUniform(material->uniforms[i]);
394     }
395     ymlDelete(&material_yml);
396     tekDeleteShaderProgram(shader_program_id);
397     tekChainThrow(tek_exception);
398 }
399
400     ymlDelete(&material_yml);
401     return SUCCESS;
402 }
403 /**
404 * Bind (use) a material, so that all subsequent draw calls
405 * will be drawn with this material.
406 * @param material The material to bind.
407 * @throws OPENGL_EXCEPTION if the material is invalid in some
408 * way.

```

```

407  */
408 exception tekBindMaterial(const TekMaterial* material) {
409     const uint shader_program_id = material->shader_program_id;
410     tekBindShaderProgram(shader_program_id); // load the
internal shader.
411     for (uint i = 0; i < material->num_uniforms; i++) { // bind
all uniforms
412         const TekMaterialUniform* uniform = material-
>uniforms[i];
413         switch (uniform->type) { // map each type to the
correct shader bind function.
414             case UINTEGER_DATA:
415                 long* uinteger = uniform->data;
416
tekChainThrow(tekShaderUniformInt(shader_program_id, uniform->name,
(int)(*uinteger)));
417                 break;
418             case UFLOAT_DATA:
419                 double* ufloat = uniform->data;
420
tekChainThrow(tekShaderUniformFloat(shader_program_id, uniform-
>name, (float)(*ufloat)));
421                 break;
422             case TEXTURE_DATA:
423                 uint* texture_id = uniform->data; // textures
also need to be bound to a slot.
424                 tekBindTexture(*texture_id, 1);
425
tekChainThrow(tekShaderUniformInt(shader_program_id, uniform->name,
1));//(int)(*texture_id));
426                 break;
427             case VEC2_DATA:
428
tekChainThrow(tekShaderUniformVec2(shader_program_id, uniform->name,
uniform->data));
429                 break;
430             case VEC3_DATA:
431
tekChainThrow(tekShaderUniformVec3(shader_program_id, uniform->name,
uniform->data));
432                 break;
433             case VEC4_DATA:

```

```

434
tekChainThrow(tekShaderUniformVec4(shader_program_id, uniform->name,
uniform->data));
435             break;
436         default:
437             break;
438         }
439     }
440     return SUCCESS;
441 }
442
443 /**
444  * Return whether a material has a specific type of uniform.
445  * Useful to check if a material has a projection matrix etc.
446  * @param material The material to check.
447  * @param uniform_type The type to check if the material has.
448  * @return 1 if the material has that type, 0 otherwise.
449 */
450 flag tekMaterialHasUniformType(TekMaterial* material, const
flag uniform_type) {
451     const TekMaterialUniform* uniform = 0;
452     // classic linear search
453     for (uint i = 0; i < material->num_uniforms; i++) {
454         const TekMaterialUniform* loop_uniform = material-
>uniforms[i];
455         if (loop_uniform->type == uniform_type) {
456             return 1; // quit early if we can
457         }
458     }
459 }
460
461 /**
462  * Template function for binding uniforms.
463  */
464 #define MATERIAL_BIND_UNIFORM_FUNC(func_name, func_type,
bind_func) \
465 exception func_name(const TekMaterial* material, func_type
uniform_data, flag uniform_type) { \
466     const TekMaterialUniform* uniform; \
467     for (uint i = 0; i < material->num_uniforms; i++) { \
468         const TekMaterialUniform* loop_uniform = material-
>uniforms[i]; \

```

```

469         if (loop_uniform->type == uniform_type) { \
470             uniform = loop_uniform; \
471             break; \
472         } \
473     } \
474     if (!uniform) \
475         tekThrow(FAILURE, "Material does not have such a \
uniform."); \
476     tekChainThrow(bind_func(material->shader_program_id, \
uniform->name, uniform_data)); \
477     return SUCCESS; \
478 } \
479 
480 MATERIAL_BIND_UNIFORM_FUNC(tekBindMaterialVec3, vec3, \
tekShaderUniformVec3); \
481 MATERIAL_BIND_UNIFORM_FUNC(tekBindMaterialMatrix, mat4, \
tekShaderUniformMat4); \
482 \
483 /**
484 * Delete a material, freeing any allocated memory.
485 * @param material The material to delete.
486 */
487 void tekDeleteMaterial(const TekMaterial* material) {
488     // delete the uniforms
489     for (uint i = 0; i < material->num_uniforms; i++)
490         tekDeleteUniform(material->uniforms[i]);
491 
492     // delete the shader program
493     tekDeleteShaderProgram(material->shader_program_id);
494 }

```

tekgl/material.h

```

1 #pragma once
2 
3 #include <cglm/mat4.h>
4 
5 #include "../tekgl.h"
6 #include "../core/exception.h"
7 
8 #define MODEL_MATRIX_DATA      -1
9 #define VIEW_MATRIX_DATA       -2
10 #define PROJECTION_MATRIX_DATA -3
11 #define CAMERA_POSITION_DATA   -4

```

```

12
13  typedef struct TekMaterialUniform {
14      char* name;
15      flag type;
16      void* data;
17 } TekMaterialUniform;
18
19  typedef struct TekMaterial {
20      uint shader_program_id;
21      uint num_uniforms;
22      TekMaterialUniform** uniforms;
23 } TekMaterial;
24
25 exception tekCreateMaterial(const char* filename, TekMaterial* material);
26 exception tekBindMaterial(const TekMaterial* material);
27 flag tekMaterialHasUniformType(TekMaterial* material, flag uniform_type);
28 exception tekBindMaterialVec3(const TekMaterial* material, vec3 uniform_data, flag uniform_type);
29 exception tekBindMaterialMatrix(const TekMaterial* material, mat4 uniform_data, flag uniform_type);
30 void tekDeleteMaterial(const TekMaterial* material);

```

tekgl/mesh.c

```

1  #include <errno.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5
6  #include "mesh.h"
7  #include "../core/list.h"
8  #include "../core/file.h"
9
10 #include "glad/glad.h"
11
12 #define READ_VERTICES 0
13 #define READ_INDICES 1
14 #define READ_LAYOUT 2
15
16 /**
17 * Create a buffer in opengl and fill it with data.

```

```

18  * @param buffer_type The type of buffer e.g. GL_ARRAY_BUFFER,
GL_ELEMENT_ARRAY_BUFFER etc.
19  * @param buffer_data The data being stored in the buffer.
20  * @param buffer_size The size in bytes of the array.
21  * @param buffer_usage The way the buffer will be used e.g.
GL_STATIC_DRAW
22  * @param buffer_id The outputted id of the buffer.
23  * @throws OPENGL_EXCEPTION if the buffer could not be created.
24 */
25 exception tekCreateBuffer(const GLenum buffer_type, const void*
buffer_data, const long buffer_size, const GLenum buffer_usage,
uint* buffer_id) {
26     if (!buffer_data) tekThrow(NULL_PTR_EXCEPTION, "Buffer data
cannot be null.");
27     if (!buffer_id) tekThrow(NULL_PTR_EXCEPTION, "Buffer id
cannot be null.");
28
29     // create and bind a new empty buffer
30     glGenBuffers(1, buffer_id); // create many buffers in an
array with length one, so just do this one array
31     glBindBuffer(buffer_type, *buffer_id);
32
33     // fill with data and check for errors
34     glBufferData(buffer_type, buffer_size, buffer_data,
buffer_usage);
35     switch (glGetError()) {
36     case GL_INVALID_ENUM:
37         tekThrow(OPENGL_EXCEPTION, "Invalid buffer type or
invalid draw type.");
38     case GL_INVALID_VALUE:
39         tekThrow(OPENGL_EXCEPTION, "Buffer size cannot be
negative.");
40     case GL_INVALID_OPERATION:
41         tekThrow(OPENGL_EXCEPTION, "Invalid operation - buffer
is not bound or is immutable.");
42     case GL_OUT_OF_MEMORY:
43         tekThrow(OPENGL_EXCEPTION, "Failed to create buffer
with size specified.")
44     default:
45         return SUCCESS;
46     }
47 }
48

```

```

49 /**
50  * Generate the vertex attributes for a mesh based on the
51  * layout array.
52  * @param layout The layout array.
53  * @param len_layout The length of the layout array.
54  * @throws OPENGL_EXCEPTION if length of the layout is 0.
55 */
56 static exception tekGenerateVertexAttributes(const int* layout,
57 const uint len_layout) {
58     // find the total size of each vertex
59     int layout_size = 0;
60     for (uint i = 0; i < len_layout; i++) layout_size += layout[i];
61     // convert size to bytes
62     layout_size *= sizeof(float);
63     // create attrib pointer for each section of the layout
64     int prev_layout = 0;
65     for (uint i = 0; i < len_layout; i++) {
66         if (layout[i] == 0)
67             tekThrow(OPENGL_EXCEPTION, "Cannot have layout of
size 0.");
68         glVertexAttribPointer(i, layout[i], GL_FLOAT, GL_FALSE,
69 layout_size, (void*)(prev_layout * sizeof(float)));
70         glEnableVertexAttribArray(i);
71         prev_layout += layout[i];
72     }
73     return SUCCESS;
74 }
75 /**
76  * Create a mesh from a collection of arrays. Vertices is the
array of actual vertex data. Indices is an array that says the order
to access the vertex array in to draw the mesh. The layout describes
the length of each chunk of data in the vertex array, e.g. a
position vec3, a normal vec3 and a vec2 texture coordinate would
have a layout of {3, 3, 2}.
77 * @param vertices An array of vertices.
78 * @param len_vertices The length of the array of vertices.
79 * @param indices An array of indices.
80 * @param len_indices The length of the array of indices.
81 * @param layout The layout array.

```

```

82     * @param len_layout The length of the layout array.
83     * @param mesh_ptr A pointer to an existing TekMesh struct
which will be filled with the data for this mesh.
84     * @throws OPENGL_EXCEPTION if could not be created.
85     */
86 exception tekCreateMesh(const float* vertices, const long
len_vertices, const uint* indices, const long len_indices, const
int* layout, const uint len_layout, TekMesh* mesh_ptr) {
87     if (!mesh_ptr) tekThrow(NULL_PTR_EXCEPTION, "Mesh pointer
cannot be null.")
88
89     // unbind the vertex array so we don't interfere with
anything else
90     glBindVertexArray(0);
91     glBindBuffer(GL_ARRAY_BUFFER, 0);
92     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
93
94     // vertex array will keep track of buffers and attribute
pointers for us
95     // create and bind the vertex array, the currently bound
vertex array will track what we do next
96     glGenVertexArrays(1, &mesh_ptr->vertex_array_id);
97     glBindVertexArray(mesh_ptr->vertex_array_id);
98
99     // create the vertex buffer and element buffer
100    // vertex buffer stores data, element buffer stores the
order to draw vertices in
101    tekChainThrow(tekCreateBuffer(
102        GL_ARRAY_BUFFER, vertices, len_vertices *
sizeof(float), GL_STATIC_DRAW, &mesh_ptr->vertex_buffer_id
103    ));
104    tekChainThrow(tekCreateBuffer(
105        GL_ELEMENT_ARRAY_BUFFER, indices, len_indices *
sizeof(uint), GL_STATIC_DRAW, &mesh_ptr->element_buffer_id
106    ));
107
108    tekChainThrow(tekGenerateVertexAttributes(layout,
len_layout));
109
110    // keep track of how many elements are in element buffer -
we need this to draw later on
111    mesh_ptr->num_elements = (int)len_indices;
112

```

```

113     return SUCCESS;
114 }
115
116 /**
117 *
118 * @param func_name The name of the function.
119 * @param func_type The return type of the function.
120 * @param conv_func The function to convert from a string to
that type(ish)
121 * @param conv_type The actual type of the conv_func, will be
casted to func_type after conversion.
122 */
123 #define STRING_CONV_FUNC(func_name, func_type, conv_func,
conv_type) \
124 exception func_name(const char* string, func_type* output) { \
125     char* check = 0; \
126     const conv_type converted = conv_func(string, &check); \
127     int error = errno; \
128     if ((string + strlen(string)) != check) || (error == \
ERANGE)) \
129         tekThrow(FAILURE, "Failed to convert string data while
processing mesh."); \
130     *output = (func_type)converted; \
131     return SUCCESS; \
132 } \
133
134 STRING_CONV_FUNC(stringToFloat, float, strtod, double);
135 STRING_CONV_FUNC(stringToInt, uint, strtold, long);
136 STRING_CONV_FUNC(stringToInt, int, strtold, long);
137
138 /**
139 * Build an array from a backwards list.
140 * @param func_name The name of the function.
141 * @param array_type The type of array to build.
142 * @param conv_func The function used to convert a string into
the type of the array.
143 */
144 #define ARRAY_BUILD_FUNC(func_name, array_type, conv_func) \
145 exception func_name(const List* list, array_type* array) { \
146     uint index = list->length - 1; \
147     const ListItem* item = list->data; \
148     while (item) { \
149         const char* string = item->data; \

```

```

150         array_type data; \
151         tekChainThrow(conv_func(string, &data)); \
152         array[index--] = (array_type) data; \
153         item = item->next; \
154     } \
155     return SUCCESS; \
156 } \
157 \
158 ARRAY_BUILD_FUNC(tekCreateMeshVertices, float, stringToFloat); \
159 ARRAY_BUILD_FUNC(tekCreateMeshIndices, uint, stringToInt); \
160 ARRAY_BUILD_FUNC(tekCreateMeshLayout, int, stringToInt); \
161 \
162 /**
163 * Convert the lists produced by tekReadMeshLists into arrays. \
Very useful because the lists are produced in reverse order and need \
to be flipped. And the lists are linked lists which cannot be used \
as arrays. \
164 * FAQ: Where is the length? Its stored in the lists already... \
165 * Does this allocate the lists for me? No, because then its \
easier to free everything if this fails. \
166 * @param vertices The list of vertices. \
167 * @param indices The list of indices. \
168 * @param layout The layout list. \
169 * @param vertex_array The outputted vertex array. \
170 * @param index_array The outputted index array. \
171 * @param layout_array The outputted layout array. \
172 * @throws FAILURE if there is an item in the list that is \
badly formatted. (still strings at this point) \
173 */ \
174 static exception tekCreateMeshArrays(const List* vertices, \
const List* indices, const List* layout, float* vertex_array, uint* \
index_array, int* layout_array) { \
175     // using template functions for converting backwards list \
to arrays. \
176     tekChainThrow(tekCreateMeshVertices(vertices, \
vertex_array)); \
177     tekChainThrow(tekCreateMeshIndices(indices, index_array)); \
178     tekChainThrow(tekCreateMeshLayout(layout, layout_array)); \
179     return SUCCESS; \
180 } \
181 \
182 /**

```

```

183  * Read a mesh file into lists. The file should be loaded into
184  * a buffer and the lists should already be created and ready to use.
185  * Should only really be used in \ref tekReadMeshArrays.
186  * IMPORTANT: for linked list speed, the items are inserted at
187  * the start not the end. So the items come out back to front. Haha. So
188  * please dont use this function anywhere else.
189  * @param buffer The loaded file.
190  * @param vertices The list of vertices.
191  * @param indices The list of indices.
192  * @param layout The layout list.
193  * @param position_layout_index The outputted position layout
194  * index.
195  * @throws LIST_EXCEPTION if could not append to one of the
196  * lists.
197  */
198 static exception tekReadMeshLists(char* buffer, List* vertices,
199 List* indices, List* layout, uint* position_layout_index) {
200     char* prev_c = buffer;
201     uint line = 1;
202     flag in_word = 0;
203     flag in_wildcard = 0;
204     if (position_layout_index) *position_layout_index = 0;
205
206     List* write_list = 0;
207     char* c;
208     for (c = buffer; *c; c++) {
209         if ((*c == '\n') || (*c == '\r')) line++; // line
210         counter
211         if (*c == '#') { // commented out lines should be
212             ignored
213             while ((*c != '\n') && (*c != '\r')) c++;
214             in_word = 0;
215             line++;
216             continue;
217         }
218         if ((*c == ' ') || (*c == 0x09) || (*c == '\n') || (*c
219 == '\r')) { // looking for word separators
220             if (in_word) {
221                 // split the string here, so any of the string
222                 functions see this as the end of the string
223                 *c = 0;
224                 // if the word up until now matches one of
225                 these, then thats important.

```

```

214          // for these three, it means update the list we
are writing to what is specified.
215          if (!strcmp(prev_c, "VERTICES")) write_list =
vertices;
216          else if (!strcmp(prev_c, "INDICES")) write_list
= indices;
217          else if (!strcmp(prev_c, "LAYOUT")) write_list
= layout;
218          // this one means that we need to write to the
position layout index
219          else if (!strcmp(prev_c,
"$POSITION_LAYOUT_INDEX")) {
220              write_list = 0;
221              in_wildcard = 1;
222          }
223          // if not a special word, then we need to add
the last thing to the write list or position layout variable.
224          else if (!in_wildcard) {
225              if (write_list)
tekChainThrow(listInsertItem(write_list, 0, prev_c)); // inserting
at 0, faster for linked list
226          } else {
227              if (position_layout_index)
tekChainThrow(stringToUInt(prev_c, position_layout_index));
228          }
229      }
230      in_word = 0;
231      continue;
232  }
233  if (!in_word) prev_c = c;
234  in_word = 1;
235 }
236 if (in_word) {
237     // if the last item is a section header, then there
would be no items below it, which is pointless and should be an
error anyway
238     // so dont bother with checking at this stage.
239     if (!in_wildcard) {
240         if (write_list)
tekChainThrow(listInsertItem(write_list, 0, prev_c)); // inserting
at 0, faster for linked list
241     } else {

```

```

242             tekChainThrow(stringToInt(prev_c,
position_layout_index));
243         }
244     }
245     return SUCCESS;
246 }
247
248 /**
249  * Read a mesh file into arrays.
250  * @param filename The filename of the mesh to read.
251  * @param vertex_array A pointer to where a new vertex array is
to be created.
252  * @param len_vertex_array The outputted length of the new
vertex array.
253  * @param index_array A pointer to where a new index array is
to be created.
254  * @param len_index_array The outputted length of the new index
array.
255  * @param layout_array A pointer to where a new layout array is
to be created.
256  * @param len_layout_array The outputted length of the new
layout array.
257  * @param position_layout_index The outputted position layout
index which can be specified in the mesh file.
258  * @throws MEMORY_EXCEPTION if malloc() fails.
259  * @throws FILE_EXCEPTION if file couldn't be read.
260 */
261 exception tekReadMeshArrays(const char* filename, float**  

vertex_array, uint* len_vertex_array, uint** index_array, uint*  

len_index_array, int** layout_array, uint* len_layout_array, uint*  

position_layout_index) {
262     // read mesh file into buffer.
263     uint file_size;
264     tekChainThrow(getFileSize(filename, &file_size));
265     char* buffer = (char*)malloc(file_size);
266     tekChainThrowThen(readFile(filename, file_size, buffer), {
267         free(buffer);
268     });
269
270     // create an empty list for each data type.
271     List vertices = {};
272     listCreate(&vertices);
273     List indices = {};

```

```

274     listCreate(&indices);
275     List layout = {};
276     listCreate(&layout);
277
278     // attempt to read mesh into those lists.
279     tekChainThrowThen(tekReadMeshLists(buffer, &vertices,
280                         &indices, &layout, position_layout_index), {
281         listDelete(&vertices);
282         listDelete(&indices);
283         listDelete(&layout);
284     });
285
286     // allocate buffers to copy over the data from each list
287     // into
288     *vertex_array = (float*)malloc(vertices.length *
289     sizeof(float));
290     if (!*vertex_array) tekThrow(MEMORY_EXCEPTION, "Failed to
291     allocate memory for vertices.");
292     *len_vertex_array = vertices.length;
293
294     *index_array = (uint*)malloc(indices.length *
295     sizeof(uint));
296     if (!*index_array) tekThrowThen(MEMORY_EXCEPTION, "Failed
297     to allocate memory for indices.", {
298         free(*vertex_array);
299     });
300     *len_index_array = indices.length;
301
302     *layout_array = (int*)malloc(layout.length * sizeof(int));
303     if (!*layout_array) tekThrowThen(MEMORY_EXCEPTION, "Failed
304     to allocate memory for layout.", {
305         free(*vertex_array);
306         free(*index_array);
307     });
308     *len_layout_array = layout.length;
309
310     // attempt to create mesh using these arrays.
311     const exception tek_exception =
312     tekCreateMeshArrays(&vertices, &indices, &layout, *vertex_array,
313     *index_array, *layout_array);
314
315     if (tek_exception) {
316         free(*vertex_array);

```

```

308         free(*index_array);
309         free(*layout_array);
310     }
311
312     // delete lists as they are no longer needed
313     listDelete(&vertices);
314     listDelete(&indices);
315     listDelete(&layout);
316
317     // free file buffer
318     free(buffer);
319
320     tekChainThrow(tek_exception);
321     return SUCCESS;
322 }
323
324 /**
325  * Read a file and write data into a mesh.
326  * @param filename The file that contains the mesh data.
327  * @param mesh_ptr A pointer to an existing but empty TekMesh
328  * struct that will have data written to it.
329  * @throws FILE_EXCEPTION if could not read the file.
330  * @throws MEMORY_EXCEPTION if malloc() fails.
331 */
331 exception tekReadMesh(const char* filename, TekMesh* mesh_ptr)
{
332     // make a bunch of variables to read into.
333     float* vertex_array = 0;
334     uint* index_array = 0;
335     int* layout_array = 0;
336     uint len_vertex_array = 0, len_index_array = 0,
len_layout_array = 0;
337
338     // read the file into variables.
339     tekChainThrow(tekReadMeshArrays(filename, &vertex_array,
&len_vertex_array, &index_array, &len_index_array, &layout_array,
&len_layout_array, NULL));
340
341     // attempt to create the mesh
342     const exception tek_exception = tekCreateMesh(vertex_array,
(len)len_vertex_array, index_array, (long)len_index_array,
layout_array, len_layout_array, mesh_ptr);
343

```

```

344     // free everything because either path that happens next
345     // doesn't want them in memory
346     free(vertex_array);
347     free(index_array);
348     free(layout_array);
349
350     // check for an exception
351     tekChainThrow(tek_exception);
352     return SUCCESS;
353 }
354 /**
355  * Draw a mesh to the screen. Requires setting a
356  * shader/material first in order to render anything.
357  * @param mesh_ptr A pointer to the mesh to be drawn.
358  */
359 void tekDrawMesh(const TekMesh* mesh_ptr) {
360     // binding vertex array buffer - OpenGL should know which
361     // vertex buffer and element buffer we want from this
362     glBindVertexArray(mesh_ptr->vertex_array_id);
363
364     // draw elements as triangles, using the number of elements
365     // we tracked before
366     glDrawElements(GL_TRIANGLES, mesh_ptr->num_elements,
367     GL_UNSIGNED_INT, 0);
368 }
369 /**
370  * Recreate an existing mesh and provide a new set of vertices,
371  * indices and layout.
372  * Can set any parameter to be NULL if you want to retain the
373  * old data for that.
374  * @param mesh_ptr The pointer to the existing mesh or NULL to
375  * not change.
376  * @param vertices The array of vertices
377  * @param len_vertices The length of the array of vertices or
378  * NULL to not change.
379  * @param indices The array of indices.
380  * @param len_indices The length of the array of indices or
381  * NULL to not change.
382  * @param layout The layout array.
383  * @param len_layout The length of the layout array.
384  * @throws OPENGL_EXCEPTION if layout has length of 0.

```

```

377  */
378 exception tekRecreateMesh(TekMesh* mesh_ptr, const float*
vertices, const long len_vertices, const uint* indices, const long
len_indices, const int* layout, const uint len_layout) {
379     glBindVertexArray(mesh_ptr->vertex_array_id);
380     if (vertices) { // update vertices if not null
381         glBindBuffer(GL_ARRAY_BUFFER, mesh_ptr-
>vertex_buffer_id);
382         glBufferData(GL_ARRAY_BUFFER, (long)(len_vertices *
sizeof(float)), vertices, GL_STATIC_DRAW);
383     }
384
385     if (indices) { // update indices if not null
386         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh_ptr-
>element_buffer_id);
387         glBufferData(GL_ELEMENT_ARRAY_BUFFER, (long)
(len_indices * sizeof(uint)), indices, GL_STATIC_DRAW);
388         mesh_ptr->num_elements = (int)len_indices;
389     }
390
391     if (layout) { // update layout if not null
392         tekChainThrow(tekGenerateVertexAttributes(layout,
len_layout));
393     }
394
395     return SUCCESS;
396 }
397
398 /**
399  * Delete a mesh by deleting any opengl buffers that were
allocated for it.
400  * @param mesh_ptr The pointer to the mesh to delete.
401  */
402 void tekDeleteMesh(const TekMesh* mesh_ptr) {
403     // glDelete...() expects an array of ids, so need to
provide pointer and length of 1 to pretend its an array.
404     glDeleteVertexArrays(1, &mesh_ptr->vertex_array_id);
405     glDeleteBuffers(1, &mesh_ptr->vertex_buffer_id);
406     glDeleteBuffers(1, &mesh_ptr->element_buffer_id);
407 }

```

tekgl/mesh.h

```
1 #pragma once
```

```

2
3 #include "../tekg1.h"
4 #include "../core/exception.h"
5
6 typedef struct TekMesh {
7     uint vertex_array_id;
8     uint vertex_buffer_id;
9     uint element_buffer_id;
10    int num_elements;
11 } TekMesh;
12
13 exception tekReadMeshArrays(const char* filename, float** vertex_array, uint* len_vertex_array, uint** index_array, uint* len_index_array, int** layout_array, uint* len_layout_array, uint* position_layout_index);
14 exception tekReadMesh(const char* filename, TekMesh* mesh_ptr);
15 exception tekCreateMesh(const float* vertices, long len_vertices, const uint* indices, long len_indices, const int* layout, uint len_layout, TekMesh* mesh_ptr);
16 exception tekRecreateMesh(TekMesh* mesh_ptr, const float* vertices, long len_vertices, const uint* indices, long len_indices, const int* layout, uint len_layout);
17 void tekDrawMesh(const TekMesh* mesh_ptr);
18 void tekDeleteMesh(const TekMesh* mesh_ptr);

```

tekg1/shader.c

```

1 #include "shader.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <glad/glad.h>
6 #include "../core/file.h"
7
8 /**
9  * Delete a shader using its id.
10 * @param shader_id The id of the shader to delete.
11 */
12 static void tekDeleteShader(const uint shader_id) {
13     if (shader_id) glDeleteShader(shader_id); // check if
14     shader is null before deleting.
15 }
16 /**

```

```

17  * Compile a piece of shader code contained within a file and
return an id of this shader.
18  * @param shader_type The type of shader, either VERTEX_SHADER,
GEOMETRY_SHADER, FRAGMENT_SHADER.
19  * @param shader_filename The filename of the shader file.
20  * @param shader_id A pointer to an unsigned integer that will
be overwritten with the shader id.
21  * @throws MEMORY_EXCEPTION if malloc() fails.
22  * @throws FILE_EXCEPTION if the file could not be read.
23  */
24 static exception tekCreateShader(const GLenum shader_type,
const char* shader_filename, uint* shader_id) {
25     // get the size of the shader file to read
26     uint file_size;
27     tekChainThrow(getFileSize(shader_filename, &file_size));
28
29     // read the file contents into a buffer of that size
30     char* file_buffer = (char*)malloc(file_size *
sizeof(char));
31     if (!file_buffer) tekThrow(MEMORY_EXCEPTION, "Failed to
allocate memory for shader.")
32     tekChainThrow(readFile(shader_filename, file_size,
file_buffer));
33
34     // opengl calls to create shader
35     *shader_id = glCreateShader(shader_type);
36
37     if (*shader_id == 0) tekThrow(OPENGL_EXCEPTION, "Failed to
create shader.");
38     if (*shader_id == GL_INVALID_ENUM)
tekThrow(OPENGL_EXCEPTION, "Invalid shader type.");
39
40     glShaderSource(*shader_id, 1, &file_buffer, NULL);
41     glCompileShader(*shader_id);
42     free(file_buffer);
43
44     // check for errors during compilation
45     int success;
46     glGetShaderiv(*shader_id, GL_COMPILE_STATUS, &success);
47     if (!success) {
48         char info_log[E_MESSAGE_SIZE];
49         glGetShaderInfoLog(*shader_id, E_MESSAGE_SIZE, NULL,
info_log);

```

```

50         tekDeleteShader(*shader_id);
51         tekThrow(OPENGL_EXCEPTION, info_log);
52     }
53
54     return SUCCESS;
55 }
56
57 /**
58  * Bind (use) a shader program using its id. All subsequent
draw calls will then use this shader.
59  * @param shader_program_id The id of the shader program to
bind.
60 */
61 void tekBindShaderProgram(const uint shader_program_id) {
62     glUseProgram(shader_program_id); // use it
63 }
64
65 /**
66  * Delete a shader program using its id.
67  * @param shader_program_id The id of the shader program to
delete.
68 */
69 void tekDeleteShaderProgram(const uint shader_program_id) {
70     glDeleteProgram(shader_program_id); // delete it idk?
71 }
72
73 /**
74  * Create a shader program from a vertex shader and fragment
shader id.
75  * @param vertex_shader_id The id of the vertex shader.
76  * @param fragment_shader_id The id of the fragment shader.
77  * @param shader_program_id A pointer to an unsigned integer
where the shader id will be written.
78  * @throws OPENGL_EXCEPTION if the shader program cannot be
created.
79 */
80 static exception tekCreateShaderProgramWiFi(const uint
vertex_shader_id, const uint fragment_shader_id, uint*
shader_program_id) {
81     if (!vertex_shader_id || !fragment_shader_id)
tekThrow(NULL_PTR_EXCEPTION, "Cannot create shader program with id
0.")
82

```

```

83     // create an empty shader program
84     *shader_program_id = glCreateProgram();
85     if (!*shader_program_id) tekThrow(OPENGL_EXCEPTION, "Failed
to create shader program.");
86
87     // attach shaders
88     glAttachShader(*shader_program_id, vertex_shader_id);
89     glAttachShader(*shader_program_id, fragment_shader_id);
90
91     // link program
92     glLinkProgram(*shader_program_id);
93
94     return SUCCESS;
95 }
96
97 /**
98  * Create a shader program from a vertex shader, geometry
99  * shader and fragment shader id.
100 * @param vertex_shader_id The id of the vertex shader.
101 * @param geometry_shader_id The id of the geometry shader.
102 * @param fragment_shader_id The id of the fragment shader.
103 * @param shader_program_id A pointer to an unsigned integer
104 where the shader id will be written.
105 * @throws OPENGL_EXCEPTION if the shader program cannot be
106 created.
107 */
108 static exception tekCreateShaderProgramViGiFi(const uint
109 vertex_shader_id, const uint geometry_shader_id, const uint
110 fragment_shader_id, uint* shader_program_id) {
111     if (!vertex_shader_id || !geometry_shader_id || !
112 fragment_shader_id) tekThrow(NULL_PTR_EXCEPTION, "Cannot create
113 shader program with id 0.")
114
115     // create an empty shader program
116     *shader_program_id = glCreateProgram();
117     if (!*shader_program_id) tekThrow(OPENGL_EXCEPTION, "Failed
to create shader program.");
118
119     // attach shaders
120     glAttachShader(*shader_program_id, vertex_shader_id);
121     glAttachShader(*shader_program_id, geometry_shader_id);
122     glAttachShader(*shader_program_id, fragment_shader_id);
123
124     // link program
125     glLinkProgram(*shader_program_id);
126
127     return SUCCESS;
128 }

```

```

117     // link program
118     glLinkProgram(*shader_program_id);
119
120     return SUCCESS;
121 }
122
123 /**
124  * Create a shader program from a vertex and fragment shader.
125  * @param vertex_shader_filename The filename of the vertex
126  * shader.
127  * @param fragment_shader_filename The filename of the fragment
128  * shader.
129  * @param shader_program_id A pointer to an unsigned integer
130  * where the shader program id will be written.
131  * @throws OPENGL_EXCEPTION if failed to create shader
132  * @throws MEMORY_EXCEPTION if malloc() fails.
133  * @throws FILE_EXCEPTION if file cannot be read.
134  */
135 exception tekCreateShaderProgramVF(const char*
vertex_shader_filename, const char* fragment_shader_filename, uint*
shader_program_id) {
136     // create vertex, geometry and fragment shaders
137     uint vertex_shader_id;
138     tekChainThrow(tekCreateShader(GL_VERTEX_SHADER,
vertex_shader_filename, &vertex_shader_id));
139
140     uint fragment_shader_id;
141     tekChainThrow(tekCreateShader(GL_FRAGMENT_SHADER,
fragment_shader_filename, &fragment_shader_id));
142
143     // create shader program
144     tekChainThrow(tekCreateShaderProgramViFi(vertex_shader_id,
fragment_shader_id, shader_program_id));
145
146     // clean up shaders as they are not needed now
147     tekDeleteShader(vertex_shader_id);
148     tekDeleteShader(fragment_shader_id);
149
150     return SUCCESS;
151 }
152 /**

```

```

151  * Create a shader program from a vertex, geometry and fragment
152  * shader.
153  * @param vertex_shader_filename The filename of the vertex
154  * shader.
155  * @param geometry_shader_filename The filename of the geometry
156  * shader.
157  * @param fragment_shader_filename The filename of the fragment
158  * shader.
159  * @param shader_program_id A pointer to an unsigned integer
160  * where the shader program id will be written.
161  * @throws OPENGL_EXCEPTION if failed to create shader
162  * @throws MEMORY_EXCEPTION if malloc() fails.
163  * @throws FILE_EXCEPTION if file cannot be read.
164  */
165 exception tekCreateShaderProgramVGF(const char*
166 vertex_shader_filename, const char* geometry_shader_filename, const
167 char* fragment_shader_filename, uint* shader_program_id) {
168     // create vertex, geometry and fragment shaders
169     uint vertex_shader_id;
170     tekChainThrow(tekCreateShader(GL_VERTEX_SHADER,
171 vertex_shader_filename, &vertex_shader_id));
172
173     uint geometry_shader_id;
174     tekChainThrow(tekCreateShader(GL_GEOMETRY_SHADER,
175 geometry_shader_filename, &geometry_shader_id));
176
177     uint fragment_shader_id;
178     tekChainThrow(tekCreateShader(GL_FRAGMENT_SHADER,
179 fragment_shader_filename, &fragment_shader_id));
180
181     // create shader program
182
183     tekChainThrow(tekCreateShaderProgramViGiFi(vertex_shader_id,
184 geometry_shader_id, fragment_shader_id, shader_program_id));
185
186     // clean up shaders as they are not needed now
187     tekDeleteShader(vertex_shader_id);
188     tekDeleteShader(geometry_shader_id);
189     tekDeleteShader(fragment_shader_id);
190
191     return SUCCESS;
192 }
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589

```

```

182 /**
183  * Get the location of a shader uniform by name.
184  * @param shader_program_id The id of the shader program to get
185  * the uniform location from.
186  * @param uniform_name The name of the uniform.
187  * @param uniform_location A pointer to an integer that will be
188  * overwritten with the location.
189  * @throws SHADER_EXCEPTION if the uniform name does not exist.
190 */
191 exception tekGetShaderUniformLocation(const uint
192 shader_program_id, const char* uniform_name, int* uniform_location)
193 {
194     *uniform_location = glGetUniformLocation(shader_program_id,
195     uniform_name); // returns location or -1 if not exist.
196     if (*uniform_location == -1) tekThrow(OPENGL_EXCEPTION,
197     "Uniform name does not correspond to a shader uniform.");
198     return SUCCESS;
199 }
200
201 /**
202  * Write an integer into a shader uniform.
203  * @param shader_program_id The id of the shader program to
204  * write to.
205  * @param uniform_name The name of the uniform to write.
206  * @param uniform_value The value of the integer to write.
207  * @throws SHADER_EXCEPTION if the uniform name does not exist.
208  */
209 exception tekShaderUniformInt(const uint shader_program_id,
210 const char* uniform_name, const int uniform_value) {
211     int uniform_location;
212
213     tekChainThrow(tekGetShaderUniformLocation(shader_program_id,
214     uniform_name, &uniform_location));
215     glUniform1i(uniform_location, uniform_value); // method for
216     writing one integer.
217     return SUCCESS;
218 }
219
220 /**
221  * Write a float into a shader uniform.
222  * @param shader_program_id The id of the shader program.
223  * @param uniform_name The name of the uniform to write into.
224  * @param uniform_value The value of the float to write.

```

```

214     * @throws SHADER_EXCEPTION if the uniform name does not exist.
215     */
216 exception tekShaderUniformFloat(const uint shader_program_id,
217     const char* uniform_name, const float uniform_value) {
218     int uniform_location;
219
220     tekChainThrow(tekGetShaderUniformLocation(shader_program_id,
221         uniform_name, &uniform_location));
222     glUniform1f(uniform_location, uniform_value); // method for
223     writing a single float into a uniform
224     return SUCCESS;
225 }
226 /**
227 * Write a 2 vector into a shader uniform.
228 * @param shader_program_id The id of the shader program to
229 update.
230 * @param uniform_name The name of the uniform
231 * @param uniform_value The 3 vector to write into the uniform.
232 * @throws SHADER_EXCEPTION if the uniform name does not exist.
233 */
234 exception tekShaderUniformVec2(const uint shader_program_id,
235     const char* uniform_name, const vec2 uniform_value) {
236     int uniform_location;
237
238     tekChainThrow(tekGetShaderUniformLocation(shader_program_id,
239         uniform_name, &uniform_location));
240     glUniform2fv(uniform_location, 1, uniform_value); // method
241     for writing 2 vectors.
242     return SUCCESS;
243 }
244 /**
245 * Write a 3 vector to a shader uniform.
246 * @param shader_program_id The id of the shader program to
247 update.
248 * @param uniform_name The name of the uniform to update.
249 * @param uniform_value The value of the 3 vector to update
250 with.
251 * @throws SHADER_EXCEPTION if the uniform name does not exist.
252 */
253 exception tekShaderUniformVec3(const uint shader_program_id,
254     const char* uniform_name, const vec3 uniform_value) {

```

```

245     int uniform_location;
246
tekChainThrow(tekGetShaderUniformLocation(shader_program_id,
uniform_name, &uniform_location));
247     glUniform3fv(uniform_location, 1, uniform_value); // method
for 3 vector uniform
248     return SUCCESS;
249 }
250
251 /**
252 * Write a shader uniform for a 4 vector.
253 * @param shader_program_id The id of the shader program to
update.
254 * @param uniform_name The name of the uniform to update.
255 * @param uniform_value The 4 vector to write into the uniform.
256 * @throws SHADER_EXCEPTION if the uniform name is not found.
257 */
258 exception tekShaderUniformVec4(const uint shader_program_id,
const char* uniform_name, const vec4 uniform_value) {
259     int uniform_location;
260
tekChainThrow(tekGetShaderUniformLocation(shader_program_id,
uniform_name, &uniform_location));
261     glUniform4fv(uniform_location, 1, uniform_value); // method
for 4 vector uniform.
262     return SUCCESS;
263 }
264
265 /**
266 * Update a shader uniform of mat4 type.
267 * @param shader_program_id The shader program to update.
268 * @param uniform_name The name of the uniform to update.
269 * @param uniform_value The 4x4 matrix to update with.
270 * @throws SHADER_EXCEPTION if the uniform is not found
271 */
272 exception tekShaderUniformMat4(const uint shader_program_id,
const char* uniform_name, const mat4 uniform_value) {
273     int uniform_location;
274
tekChainThrow(tekGetShaderUniformLocation(shader_program_id,
uniform_name, &uniform_location));
275     glUniformMatrix4fv(uniform_location, 1, GL_FALSE,
uniform_value); // method for 4x4 matrix uniform.

```

```
276     return SUCCESS;
277 }
```

tekgl/shader.h

```
1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../core/exception.h"
5 #include <cglm/mat4.h>
6
7 exception tekCreateShaderProgramVF(const char*
vertex_shader_filename, const char* fragment_shader_filename, uint*
shader_program_id);
8 exception tekCreateShaderProgramVGF(const char*
vertex_shader_filename, const char* geometry_shader_filename, const
char* fragment_shader_filename, uint* shader_program_id);
9 void tekBindShaderProgram(uint shader_program_id);
10 void tekDeleteShaderProgram(uint shader_program_id);
11 exception tekShaderUniformInt(uint shader_program_id, const
char* uniform_name, int uniform_value);
12 exception tekShaderUniformFloat(uint shader_program_id, const
char* uniform_name, float uniform_value);
13 exception tekShaderUniformVec2(uint shader_program_id, const
char* uniform_name, const vec2 uniform_value);
14 exception tekShaderUniformVec3(uint shader_program_id, const
char* uniform_name, const vec3 uniform_value);
15 exception tekShaderUniformVec4(uint shader_program_id, const
char* uniform_name, const vec4 uniform_value);
16 exception tekShaderUniformMat4(uint shader_program_id, const
char* uniform_name, const mat4 uniform_value);
```

tekgl/text.c

```
1 #include "text.h"
2
3 #include "font.h"
4 #include "shader.h"
5 #include <cglm/cam.h>
6
7 #include "manager.h"
8 #include "texture.h"
9
10 mat4 text_projection;
11 uint text_shader_program_id = 0;
```

```

12
13  /**
14   * Framebuffer callback for text code. Update a projection
matrix for use in text shader.
15   * @param width The new width of the framebuffer.
16   * @param height The new height of the framebuffer.
17   */
18 void tekTextFramebufferCallback(const int width, const int
height) {
19     // when framebuffer changes size, change our projection
matrix to match the new size
20     glm_ortho(0.0f, (float)width, (float)height, 0.0f, -1.0f,
1.0f, text_projection);
21 }
22
23 /**
24  * Callback for when opengl is loaded. Creates the shader that
allows text to be drawn and which is shared
25  * between different text being drawn.
26  * @throws OPENGL_EXCEPTION .
27  * @throws LIST_EXCEPTION .
28  */
29 exception tekGLLoadTextEngine() {
30     // if already initialised, don't do it again ;D
31     if (text_shader_program_id) return SUCCESS;
32
33     // create the text shader program
34
tekChainThrow(tekCreateShaderProgramVF("../shader/text_vertex.glvs",
"../shader/text_fragment.glfs", &text_shader_program_id));
35
36     // add a callback for when framebuffer changes size
37
tekChainThrow(tekAddFramebufferCallback(tekTextFramebufferCallback))
;
38
39     // create a projection matrix based on current size
40     int width, height;
41     tekGetWindowSize(&width, &height);
42     glm_ortho(0.0f, (float)width, (float)height, 0.0f, -1.0f,
1.0f, text_projection);
43
44     return SUCCESS;

```

```

45  }
46
47 /**
48 * Delete callback for text engine, delete allocated memory.
49 */
50 void tekDeleteTextEngine() {
51     // the only thing we need to delete is the shader program
52     // if its already been deleted, don't bother doing it
again.
53     if (text_shader_program_id) {
54         tekDeleteShaderProgram(text_shader_program_id);
55         text_shader_program_id = 0;
56     }
57 }
58
59 /**
60 * Initialisation function for loading text engine.
61 */
62 tek_init tekInitTextEngine() {
63     tekAddGLLoadFunc(tekGLLoadTextEngine); // load text engine
when opengl initialised.
64     tekAddDeleteFunc(tekDeleteTextEngine); // delete stuff when
program ends.
65 }
66
67 /**
68 * Generate a mesh that represents some lettering. Sorts out
where the vertices should go, new line spacing,
69 * texture atlas coordinates etc.
70 * @param text The text to generate a mesh for.
71 * @param len_text The length of the text to draw.
72 * @param size The size of the lettering in pixels.
73 * @param font The font to draw with.
74 * @param tek_text The TekText struct to create the mesh data
for.
75 * @param vertices The outputted vertices of the mesh.
76 * @param len_vertices_ptr The length of the vertices array
created.
77 * @param indices The outputted indices of the mesh.
78 * @param len_indices_ptr The length of the indices array
created.
79 * @throws MEMORY_EXCEPTION if malloc() fails.
80 */

```

```

81 static exception tekGenerateTextMeshData(const char* text,
82 const uint len_text, const uint size, TekBitmapFont* font, TekText* tek_text,
83 float** vertices, uint* len_vertices_ptr, uint** indices,
84 uint* len_indices_ptr) {
85     // make sure that these values are set to 0, as they are
86     // expected to be initialised in other calculations.
87     tek_text->width = 0;
88     tek_text->height = 0;
89
90     // mallocate some space for vertices of glyphs
91     const uint len_vertices = 16 * len_text;
92     *vertices = (float*)malloc(len_vertices * sizeof(float));
93     if (!*vertices) tekThrow(MEMORY_EXCEPTION, "Failed to
94     allocate memory for text vertices.");
95
96     // mallocate some more space for the indices of vertices
97     const uint len_indices = 6 * len_text;
98     *indices = (uint*)malloc(len_indices * sizeof(uint));
99     if (!*indices) tekThrow(MEMORY_EXCEPTION, "Failed to
100    allocate memory for text indices.");
101
102    *len_vertices_ptr = len_vertices;
103    *len_indices_ptr = len_indices;
104
105    // a pointer to where the glyphs should draw from
106    float x = 0;
107    float y = (float)size;
108
109    // some constants for when drawing the glyphs; saves
110    // recasting and recalculating every time
111    const float scale = (float)size / (float)font-
112    >original_size;
113    const float atlas_size = (float)font->atlas_size;
114
115    // for each character of text
116    for (size_t i = 0; i < len_text; i++) {
117        if (text[i] == '\n') {
118            tek_text->width = fmaxf(tek_text->width, x);
119            x = 0;
120            y += (float)size;
121            tek_text->height += (float)size;
122
123        }
124    }

```

```

115          // add bogus indices to the array to stop weird
buggering
116          uint index_data[] = {
117              0, 0, 0,
118              0, 0, 0
119          };
120          memcpy(*indices + i * 6, &index_data,
sizeof(index_data));
121
122          continue;
123      }
124
125      // grab the glyph we need based on ascii code
126      const TekGlyph glyph = font->glyphs[text[i]];
127
128      // get the width and height of the glyph, we use this a
couple of times
129      const float glyph_width = (float)glyph.width;
130      const float glyph_height = (float)glyph.height;
131
132      // calculate the x and y position of top-left based on
glyph data (some glyphs start in different places, for example 'I'
is higher than '.')
133      const float x_pos = x + (float)glyph.bearing_x *
scale;
134      const float y_pos = y + (float)glyph.height -
(float)glyph.bearing_y * scale;
135
136      // calculate the screen size of the glyph by scaling it
137      const float width = glyph_width * scale;
138      const float height = -glyph_height * scale;
139
140      // get the pixel coordinates of where the glyph can be
found in the texture atlas
141      const float atlas_x = (float)glyph.atlas_x;
142      const float atlas_y = (float)glyph.atlas_y;
143
144      // calculate texture coordinates u0, v0 = top left, u1,
v1 = bottom right
145      const float tex_u0 = atlas_x / atlas_size;
146      const float tex_v0 = atlas_y / atlas_size;
147      const float tex_u1 = (atlas_x + glyph_width) /
atlas_size;

```

```

148         const float tex_v1 = (atlas_y + glyph_height) /
atlas_size;
149
150         // add vertex data to the vertices array
151         float vertex_data[] = {
152             x_pos, y_pos + height, tex_u0, tex_v0,
153             x_pos, y_pos, tex_u0, tex_v1,
154             x_pos + width, y_pos, tex_u1, tex_v1,
155             x_pos + width, y_pos + height, tex_u1, tex_v0
156         };
157         memcpy(*vertices + i * 16, &vertex_data,
sizeof(vertex_data));
158
159         // add index data to indices array
160         // 0 --- 1    0 -> 1 -> 2 for the first triangle
161         // | \   |    0 -> 2 -> 3 for the second triangle
162         // |   \  |
163         // |     \ |   increment by 4 indices each time, as each
glyph has 4 vertices
164         // 3 --- 2
165         uint index_data[] = {
166             i * 4, i * 4 + 1, i * 4 + 2,
167             i * 4, i * 4 + 2, i * 4 + 3
168         };
169         memcpy(*indices + i * 6, &index_data,
sizeof(index_data));
170
171         // increment draw pointer
172         x += (float)(glyph.advance >> 6) * scale;
173     }
174
175     tek_text->width = fmaxf(tek_text->width, x);
176     tek_text->height += (float)size;
177
178     return SUCCESS;
179 }
180
181 /**
182  * Create a new TekText structure using a font, some text and a
size.
183  * @param text The actual writing to be displayed.
184  * @param size The size of the lettering.
185  * @param font The font to draw with.

```

```

186     * @param tek_text A pointer to an existing but empty TekText
187     struct that will have the text data written to it.
188     *
189     exception tekCreateText(const char* text, const uint size,
190     TekBitmapFont* font, TekText* tek_text) {
191         float* vertices = 0; uint len_vertices = 0;
192         uint* indices = 0; uint len_indices = 0;
193         // find the number of characters of text we are working
194         with
195             const size_t len_text = strlen(text);
196         tekChainThrow(tekGenerateTextMeshData(text, len_text, size,
197         font, tek_text, &vertices, &len_vertices, &indices, &len_indices));
198         // define the vertex data layout: 2 floats for position, 2
199         floats for texture coordinates
200         const int layout[] = {2, 2};
201         // create a mesh for the glyph quads and texture
202         coordinates
203         tekChainThrow(tekCreateMesh(vertices, len_vertices,
204         indices, len_indices, layout, 2, &tek_text->mesh));
205         // free m allocated memory
206         free(vertices);
207         free(indices);
208         // record the font that this text uses, so that we can
209         equip it when rendering the text
210         tek_text->font = font;
211         return SUCCESS;
212     }
213
214 /**
215     * Update a TekText struct to retain the same font, but redraw
216     with different text and/or a different size.
217     * @param tek_text The text to redraw.
218     * @param text The new text to update with.
219     * @param size The new size to update with.
220     * @throws OPENGL_EXCEPTION if could not update buffers.

```

```

220  */
221 exception tekUpdateText(TekText* tek_text, const char* text,
222 const uint size) {
223     float* vertices = 0; uint len_vertices = 0;
224     uint* indices = 0; uint len_indices = 0;
225
226     // find the number of characters of text we are working
227     with
228     const size_t len_text = strlen(text);
229
230     // recreate internal meshes
231     tekChainThrow(tekGenerateTextMeshData(text, len_text, size,
232 tek_text->font, tek_text, &vertices, &len_vertices, &indices,
233 &len_indices));
234     tekChainThrow(tekRecreateMesh(&tek_text->mesh, vertices,
235 len_vertices, indices, len_indices, 0, 0));
236
237     // free unneeded data.
238     free(vertices);
239     free(indices);
240     return SUCCESS;
241 }
242
243 /**
244 * Draw text at a point with a specific colour.
245 * @param tek_text The text to draw.
246 * @param x The x-coordinate to draw the text at.
247 * @param y The y-coordinate to draw the text at.
248 * @param colour The colour of the text (rgba)
249 * @throws OPENGL_EXCEPTION if could not draw the text.
250 * @throws SHADER_EXCEPTION if shader was not loaded.
251 */
252 exception tekDrawColouredText(const TekText* tek_text, const
253 float x, const float y, const vec4 colour) {
254     // make sure that text engine has been initialised, and
255     bind it
256     if (!text_shader_program_id) tekThrow(OPENGL_EXCEPTION, "No
257 text shader is available to use.");
258     tekBindShaderProgram(text_shader_program_id);
259
260     // bind font atlas texture
261     tekBindTexture(tek_text->font->atlas_id, 0);
262
263
264

```

```

255     // set all of our shader uniforms for drawing text
256     tekChainThrow(tekShaderUniformMat4(text_shader_program_id,
257         "projection", text_projection));
257     tekChainThrow(tekShaderUniformInt(text_shader_program_id,
258         "atlas", 0));
258     tekChainThrow(tekShaderUniformFloat(text_shader_program_id,
259         "draw_x", x));
259     tekChainThrow(tekShaderUniformFloat(text_shader_program_id,
260         "draw_y", y));
260     tekChainThrow(tekShaderUniformVec4(text_shader_program_id,
261         "text_colour", colour));
261
262     // draw mesh
263     tekDrawMesh(&tek_text->mesh);
264
265     return SUCCESS;
266 }
267
268 /**
269 * Draw text of a specific colour at a coordinate, and rotate
270 by an angle around a different point.
271 * @param tek_text The text to draw.
272 * @param x The x-coordinate of the text to draw.
273 * @param y The y-coordinate of the text to draw.
274 * @param colour The colour of the text (rgba)
275 * @param rot_x The x-coordinate to rotate the text around.
276 * @param rot_y The y-coordinate to rotate the text around.
277 * @param angle The angle to rotate the text by.
278 * @throws OPENGL_EXCEPTION if could not draw.
279 * @throws SHADER_EXCEPTION if shader not loaded.
280 */
280 exception tekDrawColouredRotatedText(const TekText* tek_text,
281 const float x, const float y, const vec4 colour, const float rot_x,
282 const float rot_y, const float angle) {
283     // make sure that text engine has been initialised, and
284 bind it
285     if (!text_shader_program_id) tekThrow(OPENGL_EXCEPTION, "No
286 text shader is available to use.");
287     tekBindShaderProgram(text_shader_program_id);
288
289     // create a rotation matrix
290     const float sin_angle = sinf(angle);
291     const float cos_angle = cosf(angle);

```

```

288
289     mat4 rotation;
290     glm_mat4_identity(rotation);
291     rotation[0][0] = cos_angle;
292     rotation[1][0] = -sin_angle;
293     rotation[3][0] = rot_x * (1 - cos_angle) + rot_y *
sin_angle;
294     rotation[0][1] = sin_angle;
295     rotation[1][1] = cos_angle;
296     rotation[3][1] = rot_y * (1 - cos_angle) - rot_x *
sin_angle;
297
298     glm_mat4_mul(text_projection, rotation, rotation);
299
300     // bind font atlas texture
301     tekBindTexture(tek_text->font->atlas_id, 0);
302
303     // set all of our shader uniforms for drawing text
304     tekChainThrow(tekShaderUniformMat4(text_shader_program_id,
"projection", rotation));
305     tekChainThrow(tekShaderUniformInt(text_shader_program_id,
"atlas", 0));
306     tekChainThrow(tekShaderUniformFloat(text_shader_program_id,
"draw_x", x));
307     tekChainThrow(tekShaderUniformFloat(text_shader_program_id,
"draw_y", y));
308     tekChainThrow(tekShaderUniformVec4(text_shader_program_id,
"text_colour", colour));
309
310     // draw mesh
311     tekDrawMesh(&tek_text->mesh);
312
313     return SUCCESS;
314 }
315
316 /**
317  * Draw some text to the screen at a position.
318  * @param tek_text The text to draw.
319  * @param x The x-coordinate to draw at.
320  * @param y The y-coordinate to draw at.
321  * @throws OPENGL_EXCEPTION if could not draw.
322 */

```

```

323 exception tekDrawText(const TekText* tek_text, const float x,
324 const float y) {
324     // archaic remnant of old code, text used to be hard coded
324     as white
325     // after adding colours, the coloured text method is used
325     but colour set to white.
326     // so calls to the old method still work the same now.
327     tekChainThrow(tekDrawColouredText(tek_text, x, y, (vec4)
327     {1.0f, 1.0f, 1.0f, 1.0f})));
328     return SUCCESS;
329 }
330
331 /**
332 * Delete a text struct, freeing any allocated memory.
333 * @param tek_text A pointer to the struct to delete.
334 */
335 void tekDeleteText(const TekText* tek_text) {
336     // mesh is the only thing that we need to delete
337     // font exists separately to text
338     tekDeleteMesh(&tek_text->mesh);
339 }
```

tekgl/text.h

```

1 #pragma once
2
3 #include <cglm/vec4.h>
4 #include "../core/exception.h"
5 #include "font.h"
6 #include "mesh.h"
7
8 typedef struct TekText {
9     float width, height;
10    TekMesh mesh;
11    TekBitmapFont* font;
12 } TekText;
13
14 exception tekCreateText(const char* text, uint size,
14 TekBitmapFont* font, TekText* tek_text);
15 exception tekUpdateText(TekText* tek_text, const char* text,
15 uint size);
16 exception tekDrawColouredText(const TekText* tek_text, float x,
16 float y, const vec4 colour);
```

```

17 exception tekDrawColouredRotatedText(const TekText* tek_text,
18 float x, float y, const vec4 colour, float rot_x, float rot_y, float
angle);
19 exception tekDrawText(const TekText* tek_text, float x, float
y);
20 void tekDeleteText(const TekText* tek_text);

```

tekgl/texture.c

```

1 #include "texture.h"
2
3 #define STB_IMAGE_IMPLEMENTATION
4 #include "../stb/stb_image.h"
5 #include "glad/glad.h"
6
7 /**
8  * Create a new texture from an image file, and return the id of
the newly created texture.
9  * It is recommended to have square textures with side lengths
that are powers of 2.
10 * e.g. 128x128, 512x512.
11 * @param filename The name of the file that should be loaded
into a texture.
12 * @param texture_id A pointer to a uint that will have the new
texture id written to it.
13 * @throws STBI_EXCEPTION if the image could not be loaded.
14 */
15 exception tekCreateTexture(const char* filename, uint*
texture_id) {
16     // load texture using stbi
17     stbi_set_flip_vertically_on_load(1);
18     int width, height, num_channels;
19     byte* image_data = stbi_load(filename, &width, &height,
&num_channels, 4);
20
21     // catch any potential errors
22     if (!image_data) tekThrow(STBI_EXCEPTION, "STBI failed to
load image.")
23
24     // create an empty image with OpenGL
25     glGenTextures(1, texture_id);
26     glBindTexture(GL_TEXTURE_2D, *texture_id);
27
28     // set texture parameters for wrapping and zooming

```

```

29     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_MIRRORED_REPEAT);
30     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_MIRRORED_REPEAT);
31     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
32     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
33
34     // send image data to OpenGL and create mipmap (different
levels of detail)
35     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, image_data);
36     glGenerateMipmap(GL_TEXTURE_2D);
37
38     stbi_image_free(image_data);
39     return SUCCESS;
40 }
41
42 /**
43 * Bind a texture to a specified texture slot, allows the
texture to be accessed in shaders.
44 * @param texture_id The id of the texture to bind
45 * @param texture_slot The slot (0, 1, 2, ... etc) to bind to.
46 */
47 void tekBindTexture(const uint texture_id, const byte
texture_slot) {
48     // set texture slot to bind texture to, then bind it
49     glBindTexture(GL_TEXTURE0 + texture_slot);
50     glBindTexture(GL_TEXTURE_2D, texture_id);
51 }
52
53 /**
54 * Delete a texture using its id using opengl.
55 * @param texture_id The id of the texture to delete.
56 */
57 void tekDeleteTexture(const uint texture_id) {
58     glDeleteTextures(1, &texture_id); // expects an array of
texture ids to delete
59     // so just point to the id and say length = 1
60 }
```

tekgl/texture.h

```
1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../core/exception.h"
5
6 exception tekCreateTexture(const char* filename, uint*
texture_id);
7 void tekBindTexture(uint texture_id, byte texture_slot);
8 void tekDeleteTexture(uint texture_id);
```

tekgui/box_manager.c

```
1 #include "box_manager.h"
2
3 #include <stdio.h>
4
5 #include "../tekgl/manager.h"
6 #include "../tekgl/shader.h"
7 #include "../core/vector.h"
8 #include <glad/glad.h>
9
10 #define NOT_INITIALISED 0
11 #define INITIALISED 1
12 #define DE_INITIALISED 2
13
14 static flag box_init = NOT_INITIALISED;
15 static flag box_gl_init = NOT_INITIALISED;
16
17 static uint box_shader = 0;
18 static uint box_vbo = 0;
19 static uint box_vao = 0;
20
21 static Vector box_mesh_buffer;
22
23 /**
24 * Delete callback for box code. Free supporting memory.
25 */
26 static void tekGuiBoxDelete() {
27     // delete opengl shenanigans
28     glDeleteVertexArrays(1, &box_vao);
29     glDeleteBuffers(1, &box_vbo);
30     tekDeleteShaderProgram(box_shader);
```

```

31
32     // delete vector
33     vectorDelete(&box_mesh_buffer);
34
35     box_init = DE_INITIALISED;
36     box_gl_init = DE_INITIALISED;
37 }
38
39 /**
40  * Callback for when opengl loads. Set up the vertex buffer +
element buffer + vertex array
41  * @throws FAILURE if not initialised.
42  * @throws OPENGL_EXCEPTION .
43  * @throws SHADER_EXCEPTION .
44 */
45 static exception tekGuiBoxGLLoad() {
46     if (box_init != INITIALISED)
47         tekThrow(FAILURE, "TekGuiBoxManager was not initialised
fully.");
48
49     // create vertex array to keep track of other buffers
50     glGenVertexArrays(1, &box_vao);
51     glBindVertexArray(box_vao);
52
53     // create vertex buffer to store vertex data
54     glGenBuffers(1, &box_vbo);
55     glBindBuffer(GL_ARRAY_BUFFER, box_vbo);
56
57     // define layout - 4 x 2 component vectors
58     const int layout[] = {
59         2, 2, 2, 2
60     };
61     const uint len_layout = sizeof(layout) / sizeof(int);
62
63     // find the total size of each vertex
64     int layout_size = 0;
65     for (uint i = 0; i < len_layout; i++) layout_size +=
layout[i];
66
67     // convert size to bytes
68     layout_size *= sizeof(float);
69
70     // create attrib pointer for each section of the layout

```

```

71     int prev_layout = 0;
72     for (uint i = 0; i < len_layout; i++) {
73         if (layout[i] == 0)
74             tekThrow(OPENGL_EXCEPTION, "Cannot have layout of
size 0.");
75         glVertexAttribPointer(i, layout[i], GL_FLOAT, GL_FALSE,
layout_size, (void*)(prev_layout * sizeof(float)));
76         glEnableVertexAttribArray(i);
77         prev_layout += layout[i];
78     }
79
80     // unbind for cleanliness
81     glBindVertexArray(0);
82     glBindBuffer(GL_ARRAY_BUFFER, 0);
83
84
tekChainThrow(tekCreateShaderProgramVGF("../shader/button.glvs",
"../shader/window.glgs", "../shader/window.glfs", &box_shader));
85
86     box_gl_init = INITIALISED;
87     return SUCCESS;
88 }
89
90 /**
91  * Initialisation function for gui boxes. Set some callbacks
and create supporting data structures.
92 */
93 tek_init tekGuiBoxInit() {
94     // callbacks
95     exception tek_exception =
tekAddGLLoadFunc(tekGuiBoxGLLoad);
96     if (tek_exception) return;
97
98     tek_exception = tekAddDeleteFunc(tekGuiBoxDelete);
99     if (tek_exception) return;
100
101    // create box buffer
102    tek_exception = vectorCreate(1, sizeof(TekGuiBoxData),
&box_mesh_buffer);
103    if (tek_exception) return;
104
105    box_init = INITIALISED;
106 }
```

```

107
108  /**
109   * Create a new box to be added to the box buffer.
110   * @param box_data A pointer to the box data to add to the
111   * buffer.
112   * @param index The outputted index of where the box data was
113   * added to the buffer.
114   * @throws FAILURE if box not initialised.
115   * @throws OPENGL_EXCEPTION if opengl not initialised.
116   */
117 exception tekGuiCreateBox(const TekGuiBoxData* box_data, uint* index) {
118     if (!box_init) tekThrow(FAILURE, "Attempted to run function
before initialised.");
119     if (!box_gl_init) tekThrow(OPENGL_EXCEPTION, "Attempted to
run function before OpenGL initialised.");
120
121     // length = index of the next item which will be added
122     *index = box_mesh_buffer.length;
123
124     // add new text button data to buffer
125     tekChainThrow(vectorAddItem(&box_mesh_buffer, box_data));
126
127     // update buffers using glBufferData to add new item
128     glBindVertexArray(box_vao);
129     glBindBuffer(GL_ARRAY_BUFFER, box_vbo);
130     glBufferData(GL_ARRAY_BUFFER, (long)(box_mesh_buffer.length
* sizeof(TekGuiBoxData)), box_mesh_buffer.internal,
GL_DYNAMIC_DRAW);
131
132     // unbind buffers for cleanliness
133     glBindVertexArray(0);
134     glBindBuffer(GL_ARRAY_BUFFER, 0);
135
136
137 /**
138  * Update a box in the box buffer with new coordinates. Will
139  * overwrite the old mesh data.
140  * @param box_data A pointer to the new box data to overwrite
into the box.
141  * @param index The index to write at in the box buffer.

```

```

141     * @throws FAILURE if not initialised.
142     * @throws OPENGL_EXCEPTION if opengl not initialised.
143     */
144 exception tekGuiUpdateBox(const TekGuiBoxData* box_data, const
145     uint index) {
146     if (!box_init) tekThrow(FAILURE, "Attempted to run function
147     before initialised.");
148     if (!box_gl_init) tekThrow(OPENGL_EXCEPTION, "Attempted to
149     run function before OpenGL initialised.");
150
151     // update the vector containing text button data
152     tekChainThrow(vectorsetItem(&box_mesh_buffer, index,
153     box_data));
154
155     // update the vertex buffer using glSubBufferData to
156     // overwrite the old data without reallocating.
157     glBindVertexArray(box_vao);
158     glBindBuffer(GL_ARRAY_BUFFER, box_vbo);
159     glBufferSubData(GL_ARRAY_BUFFER, (long)(index *
160     sizeof(TekGuiBoxData)), sizeof(TekGuiBoxData), box_data);
161
162     // unbind buffers for cleanliness
163     glBindVertexArray(0);
164     glBindBuffer(GL_ARRAY_BUFFER, 0);
165
166     return SUCCESS;
167 }
168
169 /**
170  * Draw a generic box to the screen, with a background and
171  * border colour. The box to draw is specified by
172  * the index in the buffer containing all boxes, this is given
173  * by the \ref tekGuiCreateBox method.
174  * @param index The index of the box to draw.
175  * @param background_colour The colour to fill the background
176  * with (rgba).
177  * @param border_colour The colour to fill the border with
178  * (rgba).
179  * @throws OPENGL_EXCEPTION if index out of range.
180  * @throws SHADER_EXCEPTION .
181  */
182 exception tekGuiDrawBox(const uint index, const vec4
background_colour, const vec4 border_colour) {

```

```

173     if (index >= box_mesh_buffer.length)
174         tekThrow(OPENGL_EXCEPTION, "Attempted to draw index out
of range.");
175
176     TekGuiBoxData* box_data;
177     vectorGetItemPtr(&box_mesh_buffer, index, &box_data);
178
179     // retrieve window size
180     int window_width, window_height;
181     tekGetWindowSize(&window_width, &window_height);
182
183     // bind shader and the necessary uniforms
184     tekBindShaderProgram(box_shader);
185     tekChainThrow(tekShaderUniformVec2(box_shader,
"window_size", (vec2){(float)window_width, (float)window_height}));
186     tekChainThrow(tekShaderUniformVec4(box_shader, "bg_colour",
background_colour));
187     tekChainThrow(tekShaderUniformVec4(box_shader, "bd_colour",
border_colour));
188
189     // draw the box specified by the index.
190     glBindVertexArray(box_vao);
191     glDrawArrays(GL_POINTS, (int)index, 1);
192
193     // unbind everything for cleanliness.
194     glBindVertexArray(0);
195     tekBindShaderProgram(0);
196
197     return SUCCESS;
198 }
```

tekgui/box_manager.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../core/exception.h"
5
6 #include <cglm/vec2.h>
7
8 typedef struct TekGuiBoxData {
9     vec2 minmax_x;
10    vec2 minmax_y;
11    vec2 minmax_ix;
```

```

12     vec2 minmax_iy;
13 } TekGuiBoxData;
14
15 exception tekGuiCreateBox(const TekGuiBoxData* box_data, uint*
index);
16 exception tekGuiDrawBox(uint index, const vec4
background_colour, const vec4 border_colour);
17 exception tekGuiUpdateBox(const TekGuiBoxData* box_data, uint
index);

```

tekgui/button.c

```

1 #include "button.h"
2
3 #include <stdio.h>
4
5 #include "../tekgl/manager.h"
6 #include <GLFW/glfw3.h>
7 #include "../core/list.h"
8 #include <string.h>
9
10#define NOT_INITIALISED 0
11#define INITIALISED 1
12#define DE_INITIALISED 2
13
14 static uint mouse_x = 0;
15 static uint mouse_y = 0;
16
17 static flag button_init = NOT_INITIALISED;
18
19 static List button_list = {0, 0};
20 static List button_dehover_list = {0, 0};
21
22 /**
23 * Get the index of a button in the button list by pointer.
24 * @param button The button to get the index of.
25 * @return The index of the button (possibly 0 if not in the
list)
26 */
27 static uint tekGuiGetButtonIndex(const TekGuiButton* button) {
28     const ListItem* item = 0;
29     uint index = 0;
30     // linear search.
31     // not much else to say

```

```

32     foreach(item, (&button_list), {
33         const TekGuiButton* check_ptr = (const
TekGuiButton*)item->data;
34         if (check_ptr == button) {
35             break;
36         }
37         index++;
38     });
39     return index;
40 }
41
42 /**
43 * Check if a position is within a button.
44 * @param button The button to check against.
45 * @param check_x The x coordinate to check.
46 * @param check_y The y coordinate to check.
47 * @return 1 if the button contains the coordinate, 0
otherwise.
48 */
49 static int tekGuiCheckButtonHitbox(const TekGuiButton* button,
int check_x, int check_y) {
50     // very basic maths
51     return (button->hitbox_x <= check_x) &&
(button->hitbox_x + button->hitbox_width > check_x) &&
(button->hitbox_y <= check_y) &&
(button->hitbox_y + button->hitbox_height > check_y);
55 }
56
57 /**
58 * Mouse button callback for button code. If click occurs
within a button, then send callback to the button and only that
button.
59 * @param button The button being pressed.
60 * @param action The action e.g. press or release.
61 * @param mods Any modifiers acting on the button.
62 */
63 static void tekGuiButtonMouseButtonCallback(const int button,
const int action, const int mods) {
64     const ListItem* item = 0;
65
66     // check every button until finding one that is under the
mouse.
67     // buttons in order of priority/height

```

```

68     TekGuiButton* clicked_button = 0;
69     foreach(item, (&button_list), {
70         TekGuiButton* loop_button = (TekGuiButton*)item->data;
71         if (tekGuiCheckButtonHitbox(loop_button, mouse_x,
mouse_y)) {
72             clicked_button = loop_button;
73             break;
74         }
75     });
76     if (!clicked_button) return; // no button found = not
clicking any button
77
78     // callback if exists
79     if (clicked_button->callback) {
80         TekGuiButtonCallbackData callback_data = {};
81         memset(&callback_data, 0,
sizeof(TekGuiButtonCallbackData));
82         callback_data.type =
TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK;
83         callback_data.data.mouse_button.button = button;
84         callback_data.data.mouse_button.action = action;
85         callback_data.data.mouse_button.mods = mods;
86         callback_data.mouse_x = mouse_x;
87         callback_data.mouse_y = mouse_y;
88         clicked_button->callback(clicked_button,
callback_data);
89     }
90 }
91
92 /**
93 * Mouse movement callback for button code. If mouse moves
within region of a button, the topmost button will bbe called back.
94 * @param x The x position of the mouse.
95 * @param y The y position of the mouse.
96 */
97 static void tekGuiButtonMousePosCallback(double x, double y) {
98     // convert to integers
99     mouse_x = (uint)x;
100    mouse_y = (uint)y;
101
102    // create callback data struct and set data
103    const ListItem* item = 0;
104    TekGuiButtonCallbackData callback_data = {};

```

```

105     memset(&callback_data, 0,
106     sizeof(TekGuiCallbackData));
107
108     callback_data.type = TEK_GUI_BUTTON_MOUSE_LEAVE_CALLBACK;
109     callback_data.mouse_x = mouse_x;
110     callback_data.mouse_y = mouse_y;
111
112     // check if we have just left a button we were in
113     // previously.
114     // if so, send a dehover event
115     uint index = 0;
116     item = button_dehover_list.data;
117     while (item) {
118         TekGuiButton* button = (TekGuiButton*)item->data;
119         item = item->next;
120         if (tekGuiCheckButtonHitbox(button, mouse_x, mouse_y))
121         {
122             button = NULL;
123             index++;
124             continue;
125         }
126         if (button->callback) button->callback(button,
127 callback_data);
128         listRemoveItem(&button_dehover_list, index, NULL);
129     }
130
131     // only top button should receive callback, so track if its
132     // been done yet
133     flag top_button_callbacked = 0;
134
135     // now checking for entering
136     foreach(item, (&button_list), {
137         TekGuiButton* button = (TekGuiButton*)item->data;
138         const int hitbox_check =
139         tekGuiCheckButtonHitbox(button, mouse_x, mouse_y);
140
141         // callback if touching
142         if (!top_button_callbacked && hitbox_check) {
143             callback_data.type =
144             TEK_GUI_BUTTON_MOUSE_TOUCHING_CALLBACK;
145             if (button->callback) button->callback(button,
146 callback_data);
147             top_button_callbacked = 1;

```

```

140         }
141
142         // remove button from dehover list
143         const ListItem* inner_item;
144         foreach(inner_item, (&button_dehover_list), {
145             const TekGuiButton* dehover_button = (const
TekGuiButton*)inner_item->data;
146             if (button == dehover_button) return;
147         });
148
149         // button entering check
150         if (hitbox_check) {
151             callback_data.type =
TEK_GUI_BUTTON_MOUSE_ENTER_CALLBACK;
152             listAddItem(&button_dehover_list, button);
153             if (button->callback) button->callback(button,
callback_data);
154             return;
155         }
156     });
157 }
158
159 /**
160 * Mouse scrolling callback for button code. The topmost button
will be called back if the scroll occurs in their bounds.
161 * @param x_offset The scroll amount in x-direction
162 * @param y_offset The scroll amount in y-direction
163 */
164 static void tekGuiButtonMouseScrollCallback(double x_offset,
double y_offset) {
165     // create new callback data struct
166     const ListItem* item = 0;
167     TekGuiButtonCallbackData callback_data = {};
168     memset(&callback_data, 0,
sizeof(TekGuiButtonCallbackData));
169
170     // write callback data
171     callback_data.type = TEK_GUI_BUTTON_MOUSE_SCROLL_CALLBACK;
172     callback_data.mouse_x = mouse_x;
173     callback_data.mouse_y = mouse_y;
174     callback_data.data.mouse_scroll.x_offset = x_offset;
175     callback_data.data.mouse_scroll.y_offset = y_offset;
176

```

```

177      // loop through buttons and find any which are being
touched by mouse right now
178      foreach(item, (&button_list), {
179          TekGuiButton* button = (TekGuiButton*)item->data;
180          if (tekGuiCheckButtonHitbox(button, mouse_x, mouse_y))
{
181              if (button->callback) button->callback(button,
callback_data);
182              return;
183          }
184      });
185  }
186
187 /**
188 * Delete callback for button code. Delete supporting data
structures.
189 */
190 static void tekGuiButtonDelete() {
191     // delete the lists.
192     listDelete(&button_list);
193     listDelete(&button_dehover_list);
194 }
195
196 /**
197 * Initialisation function for gui button code. Set up
callbacks
198 */
199 tek_init tekGuiButtonInit() {
200     // delete callback
201     exception tek_exception =
tekAddDeleteFunc(tekGuiButtonDelete);
202     if (tek_exception) return;
203
204     // mouse input callback
205     tek_exception =
tekAddMouseButtonCallback(tekGuiButtonMouseButtonCallback);
206     if (tek_exception) return;
207
208     tek_exception =
tekAddMousePosCallback(tekGuiButtonMousePosCallback);
209     if (tek_exception) return;
210

```

```

211     tek_exception =
tekAddMouseScrollCallback(tekGuiButtonMouseScrollCallback);
212     if (tek_exception) return;
213
214     // create lists and variables
215     listCreate(&button_list);
216     listCreate(&button_dehover_list);
217
218     button_init = INITIALISED;
219 }
220
221 /**
222 * Create a new button. A button is not rendered to the screen,
it is just an area that responds to mouse activity.
223 * @param button A pointer to the button. The pointer will be
checked at each mouse event.
224 * @throws LIST_EXCEPTION if could not add to the list of
buttons
225 */
226 exception tekGuiCreateButton(TekGuiButton* button) {
227     tekChainThrow(listAddItem(&button_list, button)); // what
it says on the tin
228     return SUCCESS;
229 }
230
231 /**
232 * Set the position on the window of a button in pixels.
233 * @param button The button to set the position of.
234 * @param x The new x-coordinate of the button.
235 * @param y The new y-coordinate of the button.
236 */
237 void tekGuiSetButtonPosition(TekGuiButton* button, int x, int
y) {
238     // update button position.
239     button->hitbox_x = x;
240     button->hitbox_y = y;
241 }
242
243 /**
244 * Set the size of a button in pixels.
245 * @param button The button to set the size of.
246 * @param width The new width of the button.
247 * @param height The new height of the button.

```

```

248  */
249 void tekGuiSetButtonSize(TekGuiButton* button, uint width, uint
height) {
250     // update it.
251     button->hitbox_width = width;
252     button->hitbox_height = height;
253 }
254
255 /**
256 * Bring a button to the front of the screen, so it is rendered
above other buttons
257 * @param button The button to bring to the front.
258 * @throws LIST_EXCEPTION if could not move button to front of
rendering order.
259 */
260 exception tekGuiBringButtonToFront(const TekGuiButton* button)
{
261     // get index and move to 0
262     const uint index = tekGuiGetButtonIndex(button);
263     tekChainThrow(listMoveItem(&button_list, index, 0));
264     return SUCCESS;
265 }
266
267 /**
268 * Delete a button, freeing any allocated memory
269 * @param button The button to delete
270 */
271 void tekGuiDeleteButton(const TekGuiButton* button) {
272     // complex simplicity (volume 1)
273     const uint remove_index = tekGuiGetButtonIndex(button);
274     listRemoveItem(&button_list, remove_index, NULL);
275 }

```

tekgui/button.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../core/exception.h"
5
6 #define TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK 0
7 #define TEK_GUI_BUTTON_MOUSE_ENTER_CALLBACK 1
8 #define TEK_GUI_BUTTON_MOUSE_LEAVE_CALLBACK 2
9 #define TEK_GUI_BUTTON_MOUSE_TOUCHING_CALLBACK 3

```

```

10 #define TEK_GUI_BUTTON_MOUSE_SCROLL_CALLBACK 4
11
12 struct TekGuiButton;
13
14 typedef struct TekGuiButtonCallbackData {
15     flag type;
16     uint mouse_x;
17     uint mouse_y;
18     union {
19         struct {
20             int button;
21             int action;
22             int mods;
23         } mouse_button;
24         struct {
25             double x_offset;
26             double y_offset;
27         } mouse_scroll;
28     } data;
29 } TekGuiButtonCallbackData;
30
31 typedef void(*TekGuiButtonCallback)(struct TekGuiButton*
button_ptr, TekGuiButtonCallbackData callback_data);
32
33 typedef struct TekGuiButton {
34     int hitbox_x;
35     int hitbox_y;
36     uint hitbox_width;
37     uint hitbox_height;
38     void* data;
39     TekGuiButtonCallback callback;
40 } TekGuiButton;
41
42 exception tekGuiCreateButton(TekGuiButton* button);
43 void tekGuiSetButtonPosition(TekGuiButton* button, int x, int
y);
44 void tekGuiSetButtonSize(TekGuiButton* button, uint width, uint
height);
45 exception tekGuiBringButtonToFront(const TekGuiButton* button);
46 void tekGuiDeleteButton(const TekGuiButton* button);

```

tekgui/list_window.c

```
1 #include "list_window.h"
```

```

2
3 #include <GLFW/glfw3.h>
4
5 #include "tekgui.h"
6
7 /**
8  * Remove a text mesh from the lookup, freeing any associated
memory with that lookup.
9  * @param window The window to remove the lookup from.
10 * @param key The key to remove.
11 * @throws HASHTABLE_EXCEPTION if key does not exist.
12 */
13 static exception tekGuiListWindowRemoveLookup(TekGuiListWindow*
window, const char* key) {
14     // get text from the hashtable
15     TekText* tek_text;
16     tekChainThrow(hashtableGet(&window->text_lookup, key,
&tek_text));
17
18     // free it
19     if (tek_text) {
20         tekDeleteText(tek_text);
21         free(tek_text);
22     }
23
24     // now remove the dead pointer
25     tekChainThrow(hashtableRemove(&window->text_lookup, key));
26     return SUCCESS;
27 }
28
29 /**
30  * Clean out the lookup table, remove any items that are no
longer in the list being displayed.
31  * @param window The window for which to clean the lookup.
32  * @throws MEMORY_EXCEPTION if malloc() fails.
33  * @throws HASHTABLE_EXCEPTION .
34 */
35 static exception tekGuiListCleanLookup(TekGuiListWindow*
window) {
36     // get all keys in the hashtable to look through them
37     char** keys;
38     tekChainThrow(hashtableGetKeys(&window->text_lookup,
&keys));

```

```

39
40     // if there is a key in the lookup that is no longer in the
list, we should remove it as it will not be rendered.
41     // nested for loop, check each lookup against each item of
the list
42     for (uint i = 0; i < window->text_lookup.num_items; i++) {
43         const char* key = keys[i];
44         const ListItem* item;
45         flag exists = 0;
46         foreach(item, window->text_list, {
47             if (!strcmp(key, item->data)) {
48                 exists = 1;
49                 break;
50             }
51         });
52         if (!exists)
tekChainThrow(tekGuiListWindowRemoveLookup(window, key));
53     }
54
55     free(keys);
56     return SUCCESS;
57 }
58
59 /**
60 * Add a text mesh to the lookup.
61 * @param window The list window to add the lookup to.
62 * @param text The string associated with that text.
63 * @param tek_text The outputted text mesh. Set to NULL if not
wanted.
64 * @throws MEMORY_EXCEPTION if could not allocate memory for
new text mesh.
65 */
66 static exception tekGuiListWindowAddLookup(TekGuiListWindow*
window, const char* text, TekText** tek_text) {
67     // allocate memory for text mesh
68     TekText* tek_text_ptr = (TekText*)malloc(sizeof(TekText));
69     if (!tek_text_ptr)
70         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for text mesh.");
71
72     // get default font and create text
73     TekBitmapFont* font;
74     tekChainThrow(tekGuiGetDefaultFont(&font));

```

```

75      tekChainThrow(tekCreateText(text, window->text_size, font,
tek_text_ptr));
76
77      // add to hash table
78      tekChainThrow(hashtableSet(&window->text_lookup, text,
tek_text_ptr));
79
80      // if wanted return the text mesh.
81      if (tek_text)
82          *tek_text = tek_text_ptr;
83      return SUCCESS;
84  }
85
86 /**
87  * Get a text mesh by looking up the string it is associated
with. If there is no such text, create the text and add it to the
lookup.
88  * @param window The window that contains the string to get.
89  * @param text The
90  * @param tek_text
91  * @throws HASHTABLE_EXCEPTION .
92  */
93 static exception tekGuiListWindowGetLookup(TekGuiListWindow*
window, const char* text, TekText** tek_text) {
94     // check if the flag is there
95     const flag has_text = hashtableHasKey(&window->text_lookup,
text);
96
97     // react appropriately
98     if (has_text) {
99         tekChainThrow(hashtableGet(&window->text_lookup, text,
tek_text));
100    } else {
101        tekChainThrow(tekGuiListWindowAddLookup(window, text,
tek_text));
102        tekChainThrow(tekGuiListCleanLookup(window));
103    }
104    return SUCCESS;
105 }
106
107 /**
108  * Delete the text mesh lookup table, freeing any allocated
memory.

```

```

109 * @param window
110 * @throws HASHTABLE_EXCEPTION .
111 */
112 static exception tekGuiListWindowDeleteLookup(TekGuiListWindow*
window) {
113     // delete all the keys
114     char** keys;
115     tekChainThrow(hashtableGetKeys(&window->text_lookup,
&keys));
116
117     for (uint i = 0; i < window->text_lookup.num_items; i++) {
118         const char* key = keys[i];
119         tekChainThrow(tekGuiListWindowRemoveLookup(window,
key));
120     }
121
122     // free everything else.
123     free(keys);
124     hashtableDelete(&window->text_lookup);
125     return SUCCESS;
126 }
127
128 /**
129 * Formula to calculate a brightness value for a colour.
130 * @param colour The colour to get the brightness of.
131 * @return The brightness between 0.0 and 1.0
132 */
133 static float tekGuiGetColourBrightness(const vec4 colour) {
134     // presumably based on human eye cones and which ones are
more sensitive to light?
135     return colour[0] * 0.299f + colour[1] * 0.587f + colour[2]
* 0.114f;
136 }
137
138 /**
139 * Unused method to modify the brightness of a colour. Will
adjust the colour in the most suitable direction to have the largest
impact on making it look different to before.
140 * @param original_colour The original colour.
141 * @param final_colour The outputted colour with different
brightness.
142 * @param delta How much to change the brightness by.
143 */

```

```

144 static void tekGuiModifyColourBrightness(vec4 original_colour,
vec4 final_colour, const float delta) {
145     // get brightness
146     const float brightness =
tekGuiGetColourBrightness(original_colour);
147
148     // if already dark, make brighter
149     if (brightness < 0.5f) {
150         glm_vec4_copy((vec4){
151             original_colour[0] + delta,
152             original_colour[1] + delta,
153             original_colour[2] + delta,
154             original_colour[3],
155         },
156         final_colour
157     );
158     // if already bright, make darker
159     } else {
160         glm_vec4_copy((vec4){
161             original_colour[0] - delta,
162             original_colour[1] - delta,
163             original_colour[2] - delta,
164             original_colour[3],
165         },
166         final_colour
167     );
168 }
169 }
170
171 /**
172 * Draw a list window, drawing each string that is in the list
and is visible.
173 * @param window The list window to draw.
174 * @throws OPENGL_EXCEPTION .
175 */
176 static exception tekGuiDrawListWindow(TekGuiWindow* window) {
177     // get list window from window data.
178     TekGuiListWindow* list_window = (TekGuiListWindow*)window-
>data;
179
180     // move button in case the window has moved
181     tekGuiSetButtonPosition(&list_window->button, list_window-
>>window.x_pos, list_window->window.y_pos);

```

```

182     tekGuiSetButtonSize(&list_window->button, list_window-
183     >window.width, list_window->window.height);
184
185     // iterate over text list
186     const ListItem* item;
187     // x has slight offset from left edge
188     const float x = (float)(list_window->window.x_pos + (int)
189     (list_window->text_size / 2));
190     float y = (float)list_window->window.y_pos; // y increments
191     for each new item
192     uint index = 0;
193     foreach(item, list_window->text_list, {
194         // firstly, skip through the list until we reach the
195         first visible item.
196         if (index < list_window->draw_index) {
197             index++;
198             item = item->next;
199             continue;
200         }
201         // if we have reached the end of the visible portion,
202         stop rendering.
203         if (index >= list_window->draw_index + list_window-
204         >num_visible) break;
205
206         // retrieve the text from the list, and look up the
207         text mesh
208         const char* text = item->data;
209         TekText* tek_text = 0;
210         tekChainThrow(tekGuiListWindowGetLookup(window, text,
211         &tek_text));
212
213         // if this is the selected or hovered item, the text
214         colour is different.
215         vec4 text_colour;
216         if (index == list_window->select_index) {
217             tekGuiModifyColourBrightness(list_window-
218             >text_colour, text_colour, 0.4f);
219             } else if (index == list_window->hover_index) {
220                 tekGuiModifyColourBrightness(list_window-
221                 >text_colour, text_colour, 0.2f);
222             } else {
223                 glm_vec4_copy(list_window->text_colour,
224                 text_colour);

```

```

213         }
214
215         tekChainThrow(tekDrawColouredText(tek_text, x, y,
text_colour));
216
217         // increment y
218         y += (float)list_window->text_size * 1.25f;
219         index++;
220     });
221     return SUCCESS;
222 }
223
224 /**
225  * Select a list window and bring it to the front of the gui.
226  * @param window The window to bring to the front.
227  * @throws LIST_EXCEPTION if could not move button in list.
228 */
229 static exception tekGuiSelectListWindow(TekGuiWindow* window) {
230     // get window to get button to move forward.
231     TekGuiListWindow* list_window = (TekGuiListWindow*)window-
>data;
232     tekChainThrow(tekGuiBringButtonToFront(&list_window-
>button));
233     return SUCCESS;
234 }
235
236 /**
237  * Get the current index being hovered / clicked based on y
coordinate of mouse.
238  * @param window The window to check against.
239  * @param mouse_y The y-coordinate of the mouse.
240  * @return The index in the list that is being hovered.
241 */
242 static int tekGuiListWindowGetIndex(TekGuiListWindow* window,
int mouse_y) {
243     // setting up for a linear interpolation type thing
244     // basically, get min and max, then see how far between min
and mouse_y
245     // then get the percentage of how far between min and max
we are.
246     // then multiply by number of visible items and add how far
scrolled. simple.
247     const int

```

```

248         min_y = window->window.y_pos,
249         max_y = min_y + window->window.height;
250
251         const float depth = (float)(mouse_y - min_y) / (float)
(max_y - min_y);
252         const int screen_index = (int)(depth * (float)window-
>num_visible);
253
254         // if the index would suggest a selection off the screen,
disallow it.
255         if (screen_index < 0 || screen_index >= window-
>num_visible)
256             return -1;
257
258         return screen_index + window->draw_index;
259     }
260
261 /**
262 * Callback for when the active area of a list window is
clicked. Will pass on the clicked index to further callbacks.
263 * @param button The internal button of the list window.
264 * @param callback_data The callback data associated with this
click. (button type, press/release)
265 */
266 static void tekGuiListWindowButtonCallback(TekGuiButton*
button, TekGuiButtonCallbackData callback_data) {
267     // get list window from button data.
268     TekGuiListWindow* window = (TekGuiListWindow*)button->data;
269     if (!window->window.visible) // invisible windows should
not be clickable.
270         return;
271     switch (callback_data.type) {
272         // hovered items should be rendered differently.
273         case TEK_GUI_BUTTON_MOUSE_TOUCHING_CALLBACK:
274             window->hover_index = tekGuiListWindowGetIndex(window,
(int)callback_data.mouse_y);
275             break;
276         // if clicked
277         case TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK:
278             // ensure that left click + release
279             if (callback_data.data.mouse_button.button !=
GLFW_MOUSE_BUTTON_LEFT || callback_data.data.mouse_button.action !=
GLFW_RELEASE)

```

```

280         break;
281         // update selection and pass on callback
282         window->select_index = tekGuiListWindowGetIndex(window,
283             (int)callback_data.mouse_y);
283         if (window->callback)
284             window->callback(window);
285         break;
286     // if scrolled
287     case TEK_GUI_BUTTON_MOUSE_SCROLL_CALLBACK:
288         // increment draw index based on scroll direction
289         if (callback_data.data.mouse_scroll.y_offset > 0.0) {
290             if (window->draw_index > 0)
291                 window->draw_index--;
292         } else if (callback_data.data.mouse_scroll.y_offset <
293             0.0) {
294             if (window->draw_index < (int)window->text_list-
295                 >length - (int)window->num_visible)
296                 window->draw_index++;
297         }
298         window->hover_index = tekGuiListWindowGetIndex(window,
299             (int)callback_data.mouse_y);
300         break;
301     default:
302         break;
303     }
304 /**
305 * Create a list window gui element, which displays a
306 scrollable list of strings.
307 * The window will automatically adjust if new strings are
308 added to the list.
309 * @param window The outputted window that is being created.
310 * @param text_list The list of strings that should be
311 displayed.
312 * @throws MEMORY_EXCEPTION if malloc() fails.
313 */
314 exception tekGuiCreateListWindow(TekGuiListWindow* window,
315 List* text_list) {
316     // load list window defaults
317     struct TekGuiListWindowDefaults list_window_defaults = {};
318
319     tekChainThrow(tekGuiGetListWindowDefaults(&list_window_defaults));

```

```

314
315     // create base window
316     tekChainThrow(tekGuiCreateWindow(&window->window));
317     window->text_list = text_list;
318     window->text_size = list_window_defaults.text_size;
319     window->num_visible = list_window_defaults.num_visible;
320     glm_vec4_copy(list_window_defaults.text_colour, window-
321     >text_colour);
321     window->window.data = window;
322     window->window.draw_callback = tekGuiDrawListWindow;
323     window->window.select_callback = tekGuiSelectListWindow;
324
325     tekGuiSetWindowSize(&window->window, window->window.width,
326     window->num_visible * window->text_size * 5 / 4);
326
327     // set up text hashtable, map strings to text objects
328     // also allows text to be reused.
329     tekChainThrow(hashtableCreate(&window->text_lookup, 8));
330     tekChainThrow(tekGuiCreateButton(&window->button));
331     tekGuiSetButtonPosition(&window->button, window-
332     >window.x_pos, window->window.y_pos);
333     tekGuiSetButtonSize(&window->button, window->window.width,
334     window->window.height);
335     window->button.callback = tekGuiListWindowButtonCallback;
336     window->button.data = window;
337 }
338
339 /**
340  * Delete a list window, freeing any memory allocated to the
341  * structure.
342  * @param window The list window to delete.
343  */
344 void tekGuiDeleteListWindow(TekGuiListWindow* window) {
345     // delete base window
346     tekGuiDeleteWindow(&window->window);
347
348     // delete text storage.
349     tekGuiListWindowDeleteLookup(window);
350 }
```

tekgui/list_window.h

```
1 #pragma once
2
3 #include "window.h"
4 #include "../core/list.h"
5 #include "../core/hashtable.h"
6 #include <cglm/vec4.h>
7
8 struct TekGuiListWindow;
9
10 typedef void (*TekGuiListWindowCallback)(struct
11 TekGuiListWindow* window);
12
13 typedef struct TekGuiListWindow {
14     TekGuiWindow window;
15     List* text_list;
16     HashTable text_lookup;
17     uint text_size;
18     vec4 text_colour;
19     uint num_visible;
20     int draw_index;
21     int hover_index;
22     int select_index;
23     TekGuiButton button;
24     void* data;
25     TekGuiListWindowCallback callback;
26 } TekGuiListWindow;
27
28 exception tekGuiCreateListWindow(TekGuiListWindow* window, List*
text_list);
29 void tekGuiDeleteListWindow(TekGuiListWindow* window);
```

tekgui/options.yml

```
1 window_defaults:
2     x_pos: 200
3     y_pos: 200
4     width: 320
5     height: 180
6     title_width: 18
7     border_width: 3
8     background_colour:
9         r: 0.7
```

```
10      g: 0.7
11      b: 0.7
12      a: 0.7
13  border_colour:
14      r: 1.0
15      g: 1.0
16      b: 1.0
17      a: 1.0
18  title_colour:
19      r: 0.0
20      g: 0.0
21      b: 0.0
22      a: 1.0
23  title: "New Window"
24 list_window_defaults:
25  text_size: 16
26  text_colour:
27      r: 0.0
28      g: 0.0
29      b: 0.0
30      a: 1.0
31  num_visible: 10
32 text_button_defaults:
33  x_pos: 200
34  y_pos: 200
35  width: 120
36  height: 30
37  border_width: 1
38  text_height: 16
39  background_colour:
40      r: 0.4
41      g: 0.8
42      b: 0.8
43      a: 1.0
44  selected_colour:
45      r: 0.8
46      g: 1.0
47      b: 1.0
48      a: 1.0
49  border_colour:
50      r: 1.0
51      g: 1.0
52      b: 1.0
```

```

53      a: 1.0
54  text_input_defaults:
55      x_pos: 200
56      y_pos: 200
57      width: 200
58      text_height: 16
59      border_width: 1
60      background_colour:
61          r: 0.4
62          g: 0.8
63          b: 0.8
64          a: 1.0
65      border_colour:
66          r: 0.8
67          g: 1.0
68          b: 1.0
69          a: 1.0
70      text_colour:
71          r: 0.0
72          g: 0.0
73          b: 0.0
74          a: 1.0

```

tekgui/option_window.c

```

1 #include "option_window.h"
2
3 #include <errno.h>
4 #include <cglm/vec3.h>
5
6 #include "../core/yml.h"
7 #include "tekgui.h"
8 #include <glad/glad.h>
9 #include <GLFW/glfw3.h>
10 #include <float.h>
11
12 #include "../core/priorityqueue.h"
13
14 /// Struct containing data about the window rendering.
15 struct TekGuiOptionsWindowDefaults {
16     const char* title;
17     uint x_pos;
18     uint y_pos;
19     uint width;

```

```

20     uint height;
21     uint text_height;
22     uint input_width;
23 };
24
25 // Struct containing data about an input option in the
window.
26 struct TekGuiOptionsWindowOption {
27     char* name;
28     char* label;
29     flag type;
30 };
31
32 // Struct containing the pointer of the option window and the
option display. Used in callbacks.
33 struct TekGuiOptionPair {
34     TekGuiOptionWindow* window;
35     TekGuiOption* option;
36 };
37
38 /**
39 * Load a single unsigned integer given a key and a yml file.
40 * @param[in] yml_file The yml file to load the uint from.
41 * @param[in] key The string key that specifies the uint.
42 * @param[out] uint_ptr A pointer to where the uint should be
written to.
43 * @throws YML_EXCEPTION if the key does not exist.
44 */
45 static exception tekGuiLoadOptionsUint(YmlFile* yml_file,
const char* key, uint* uint_ptr) {
46     // empty variables to write into.
47     YmlData* yml_data;
48     long yml_integer;
49
50     // collect data from yml files
51     tekChainThrow(ymlGet(yml_file, &yml_data, key));
52     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
53
54     // cast long to uint and write to output
55     *uint_ptr = (uint)yml_integer;
56
57     return SUCCESS;
58 }

```

```

59
60  /**
61   * Load a string from a yml file given a key.
62   * @param[in] yml_file The yml file to read from
63   * @param[in] key The key from which to get the string.
64   * @param[out] string A pointer to a char pointer which will
be set to a pointer to a new buffer containing the string.
65   * @throws YML_EXCEPTION if the yml key does not exist.
66   * @throws MEMORY_EXCEPTION if the string buffer could not be
allocated.
67   * @note This function allocates memory which needs to be
freed.
68  */
69 static exception tekGuiLoadOptionsString(YmlFile* yml_file,
const char* key, char** string) {
70     // empty variable to write into
71     YmlData* yml_data;
72
73     // collect data from yml file
74     tekChainThrow(ymlGet(yml_file, &yml_data, key));
75     tekChainThrow(ymlDataToString(yml_data, string));
76
77     return SUCCESS;
78 }
79
80 /**
81  * Get the type of user input from a string.
82  * @param[in] option_type The string to get the option type
from, of the form "$tek_..._input"
83  * @return The input type, which will be -1 A.K.A.
TEK_UNKNOWN_INPUT if the string was invalid.
84  */
85 static flag tekGuiGetOptionInputType(const char* option_type)
{
86     // different wildcards that can appear in the yml file
description
87     // using strcmp to check if match
88     // if so, return that its that type
89     if (!strcmp(option_type, "$tek_label"))
90         return TEK_LABEL;
91
92     if (!strcmp(option_type, "$tek_string_input"))
93         return TEK_STRING_INPUT;

```

```

94
95     if (!strcmp(option_type, "$tek_number_input"))
96         return TEK_NUMBER_INPUT;
97
98     if (!strcmp(option_type, "$tek_boolean_input"))
99         return TEK_BOOLEAN_INPUT;
100
101    if (!strcmp(option_type, "$tek_vec3_input"))
102        return TEK_VEC3_INPUT;
103
104    if (!strcmp(option_type, "$tek_vec4_input"))
105        return TEK_VEC4_INPUT;
106
107    if (!strcmp(option_type, "$tek_button_input"))
108        return TEK_BUTTON_INPUT;
109
110    return TEK_UNKNOWN_INPUT;
111 }
112
113 /**
114  * Get the index of an option. The index is a ranking of how
far down vertically an option should appear. Kinda like how
115  * line numbers work in BASIC.
116  * @param yml_file The yml file to read the index from
117  * @param key The name of the option
118  * @param index A pointer to where the index should be written
119  * @throws YML_EXCEPTION if option does not exist
120 */
121 static exception tekGuiLoadOptionIndex(YmlFile* yml_file,
const char* key, double* index) {
122     // empty variables to write into.
123     YmlData* yml_data;
124
125     // if no mention of an index, we can put it at the end.
126     if (ymlGet(yml_file, &yml_data, "options", key, "index") !
= SUCCESS) {
127         *index = DBL_MAX;
128         return SUCCESS;
129     }
130
131     // otherwise, attempt to read the index that is wanted.
132     tekChainThrow(ymlDataToFloat(yml_data, index));
133     return SUCCESS;

```

```

134 }
135 /**
136  * Load a single option from a yml file given the name of the
137  * input.
138  * @param[in] yml_file The yml file to read from.
139  * @param[in] key The key that the option is stored under.
140  * @param[out] option A pointer to a struct which will be
141  * filled with the data of this option.
142  * @throws YML_EXCEPTION if the key does not exist.
143  * @throws MEMORY_EXCEPTION if malloc() fails.
144  * @note This function will allocate memory that needs to be
145 freed.
146 */
147 static exception tekGuiLoadOption(YmlFile* yml_file, char*
key, struct TekGuiOptionsWindowOption* option) {
148     // empty variable to store yml stuff
149     YmlData* yml_data;
150
151     // allows user to access this option later using the name
152     // it was created with
153     option->name = key;
154
155     // get the label as a string
156     tekChainThrow(ymlGet(yml_file, &yml_data, "options", key,
157 "label"));
158     tekChainThrow(ymlDataToString(yml_data, &option->label));
159
160     // get the type as a string
161     char* type_string;
162     tekChainThrow(ymlGet(yml_file, &yml_data, "options", key,
163 "type"));
164     tekChainThrow(ymlDataToString(yml_data, &type_string));
165
166     // convert string to flag and free the string as it's no
167     // longer needed.
168     option->type = tekGuiGetOptionInputType(type_string);
169     free(type_string);
170
171     // unknown input should not be accepted
172     if (option->type == TEK_UNKNOWN_INPUT)
173         tekThrow(YML_EXCEPTION, "Unknown input type for option
window.");

```

```

168
169     return SUCCESS;
170 }
171 /**
172  * Load all options as described in a yml file.
173  * @param[in] yml_file The yml file to read from.
174  * @param[in] keys An array of keys that specify the options
175  * to load
176  * @param[in] num_keys The number of keys in the keys array
177  * @param[out] options A pointer to a pointer that will be set
178  * to a new buffer containing the loaded options.
179  * @throws MEMORY_EXCEPTION if malloc() fails
180  * @throws YML_EXCEPTION if any of the keys do not exist in
181  * the yml file.
182 */
183 static exception tekGuiLoadOptions(YmlFile* yml_file, char** keys, uint num_keys, struct TekGuiOptionsWindowOption** options) {
184     // mallocate sum mem
185     *options = (struct
186         TekGuiOptionsWindowOption*)malloc(num_keys * sizeof(struct
187             TekGuiOptionsWindowOption));
188     if (!(*options))
189         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
190 for options buffer.");
191     // order the keys in the order specified by the indices of
192     // each option.
193     PriorityQueue priority_queue;
194     priorityQueueCreate(&priority_queue);
195     for (uint i = 0; i < num_keys; i++) {
196         double option_index;
197         tekChainThrow(tekGuiLoadOptionIndex(yml_file, keys[i],
198             &option_index));
199         tekChainThrow(priorityQueueEnqueue(&priority_queue,
200             option_index, keys[i]));
201     }
202     // for each key, load the option and add it to the array.
203     const PriorityQueueItem* item = priority_queue.queue;
204     uint i = 0;
205     while (item) {

```

```

200         tekChainThrowThen(tekGuiLoadOption(yml_file, item-
>data, *options + i), {
201             // on failure, we need to free the buffer.
202             free(*options);
203             *options = 0;
204             priorityQueueDelete(&priority_queue);
205         });
206         item = item->next;
207         i++;
208     }
209
210     priorityQueueDelete(&priority_queue);
211     return SUCCESS;
212 }
213
214 /**
215  * Load all data for an options window including window
metadata from a yml file.
216  * @param[in] yml_file The yml file to load the data from.
217  * @param[out] defaults The window defaults struct containing
generic window metadata.
218  * @param[out] options An array of options data containing
each option.
219  * @param[out] len_options A pointer to where the length of
the new options array can be written.
220  * @throws MEMORY_EXCEPTION if malloc() fails.
221  * @throws YML_EXCEPTION if the yml file is malformed.
222  * @note The options array will be allocated by the function
and needs to be freed.
223 */
224 static exception tekGuiLoadOptionsYml(YmlFile* yml_file,
struct TekGuiOptionsWindowDefaults* defaults, struct
TekGuiOptionsWindowOption** options, uint* len_options) {
225     *len_options = 0;
226
227     // load window defaults by name
228     tekChainThrow(tekGuiLoadOptionsString(yml_file, "title",
&defaults->title));
229     tekChainThrow(tekGuiLoadOptionsUint(yml_file, "x_pos",
&defaults->x_pos));
230     tekChainThrow(tekGuiLoadOptionsUint(yml_file, "y_pos",
&defaults->y_pos));

```

```

231     tekChainThrow(tekGuiLoadOptionsUInt(yml_file, "width",
232     &defaults->width));
233     tekChainThrow(tekGuiLoadOptionsUInt(yml_file, "height",
234     &defaults->height));
235     tekChainThrow(tekGuiLoadOptionsUInt(yml_file,
236     "text_height", &defaults->text_height));
237     tekChainThrow(tekGuiLoadOptionsUInt(yml_file,
238     "input_width", &defaults->input_width));
239
240     // load in the array of keys
241     char** keys;
242     uint num_keys = 0;
243     tekChainThrow(ymlGetKeys(yml_file, &keys, &num_keys,
244     "options"));
245
246     // load options and update length if no errors.
247     tekChainThrow(tekGuiLoadOptions(yml_file, keys, num_keys,
248     options));
249     *len_options = num_keys;
250
251     return SUCCESS;
252 }
253 /**
254  * Read a string from a named option. Only provides a
255  * reference to the string, don't edit directly!
256  * @param window The window to read from.
257  * @param key The name of the option to read.
258  * @param string A pointer to where the char pointer should be
259  * written.
260  * @throws HASHTABLE_EXCEPTION if key does not exist.
261  */
262 exception tekGuiReadStringOption(TekGuiOptionWindow* window,
263 const char* key, char** string) {
264     // get option from hashtable and copy contents into string
265     output
266     // direct pointer into my data structure !! pls no hackkj
267     TekGuiOptionData* option_data;
268     tekChainThrow(hashtableGet(&window->option_data, key,
269     (void**)&option_data));
270     *string = option_data->data.string;
271     return SUCCESS;
272 }
```

```

263
264  /**
265   * Read a floating point (double) number from a named option.
266   * @param window The window to read from.
267   * @param key The name of the option to read.
268   * @param number A pointer to where the number is to be
written.
269   * @throws HASHTABLE_EXCEPTION if key does not exist.
270  */
271 exception tekGuiReadNumberOption(TekGuiOptionWindow* window,
const char* key, double* number) {
272     // get option from hashtable and copy contents into number
output
273     TekGuiOptionData* option_data;
274     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
275     *number = option_data->data.number;
276     return SUCCESS;
277 }
278
279 /**
280  * Read a boolean option from a named option. Will return
either 0 or 1.
281  * @param window The window to read from.
282  * @param key The name of the option to read.
283  * @param boolean A pointer to where the value is to be
written.
284  * @throws HASHTABLE_EXCEPTION if key does not exist.
285  */
286 exception tekGuiReadBooleanOption(TekGuiOptionWindow* window,
const char* key, flag* boolean) {
287     // get option from hashtable and copy contents into
boolean output
288     TekGuiOptionData* option_data;
289     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
290     *boolean = option_data->data.boolean;
291     return SUCCESS;
292 }
293
294 /**
295  * Read a vec3 from a named option. Requires a vec3 to write
into.

```

```

296     * @param window The window to read from.
297     * @param key The name of the option to read.
298     * @param vector The vector to write into.
299     * @throws HASHTABLE_EXCEPTION if key does not exist.
300 */
301 exception tekGuiReadVec3Option(TekGuiOptionWindow* window,
const char* key, vec3 vector) {
302     // get option from hashtable and copy contents into vec3
output
303     TekGuiOptionData* option_data;
304     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
305     glm_vec3_copy(option_data->data.vector_3, vector);
306     return SUCCESS;
307 }
308
309 /**
310     * Read a vec4 from a named option. Requires the vec4 to exist
to write into.
311     * @param window The window to write into
312     * @param key The name of the option to read.
313     * @param vector The vector to write into.
314     * @throws HASHTABLE_EXCEPTION if key does not exist.
315 */
316 exception tekGuiReadVec4Option(TekGuiOptionWindow* window,
const char* key, vec4 vector) {
317     // get option from hashtable and copy contents into output
318     TekGuiOptionData* option_data;
319     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
320     glm_vec3_copy(option_data->data.vector_4, vector);
321     return SUCCESS;
322 }
323
324 /**
325     * Update the text displayed by an input option based on its
type. For example, numbers need to be converted to strings
326     * in order to be displayed and such.
327     * @param window The window that contains the option
328     * @param option The option to be updated
329     * @throws HASHTABLE_EXCEPTION if option doesn't exist
330 */

```

```

331 static exception
tekGuiUpdateOptionInputText(TekGuiOptionWindow* window,
TekGuiOption* option) {
332     // for strings, we already have the text to write direct
into text.
333     if (option->type == TEK_STRING_INPUT) {
334         char* string;
335         tekChainThrow(tekGuiReadStringOption(window, option-
>display.input.name, &string));
336         tekChainThrow(tekGuiSetTextInputText(&option-
>display.input.text_input, string));
337         return SUCCESS;
338     }
339
340     // for other ones, gotta convert into a string first.
341     const uint len_buffer = 128;
342     char buffer[len_buffer];
343
344     // some values that will be filled in if its that type
345     double number;
346     flag boolean;
347     vec4 vector;
348     switch (option->type) {
349     case TEK_NUMBER_INPUT:
350         tekChainThrow(tekGuiReadNumberOption(window, option-
>display.input.name, &number));
351         sprintf(buffer, len_buffer, "% .5f", number);
352         break;
353     case TEK_BOOLEAN_INPUT:
354         tekChainThrow(tekGuiReadBooleanOption(window, option-
>display.input.name, &boolean));
355         if (boolean) {
356             sprintf(buffer, len_buffer, "True");
357         } else {
358             sprintf(buffer, len_buffer, "False");
359         }
360         break;
361     // for vectors, each option only links to one component.
so treat that component as if its just one number (cuz it is)
362     case TEK_VEC3_INPUT:
363         tekChainThrow(tekGuiReadVec3Option(window, option-
>display.input.name, vector));

```

```

364         sprintf(buffer, len_buffer, "%.5f", vector[option-
>display.input.index]);
365         break;
366     case TEK_VEC4_INPUT:
367         tekChainThrow(tekGuiReadVec4Option(window, option-
>display.input.name, vector));
368         sprintf(buffer, len_buffer, "%.5f", vector[option-
>display.input.index]);
369         break;
370     default:
371         tekThrow(FAILURE, "Unknown input type.");
372     }
373
374     // finally, update the displayed text in the option input.
375     tekChainThrow(tekGuiSetTextInputText(&option-
>display.text_input, buffer));
376
377     return SUCCESS;
378 }
379
380 /**
381 * Return whether the option is an option input that can be
written into.
382 * @param option The option to check
383 * @return 0 if not an input, 1 otherwise.
384 */
385 static flag tekGuiIsOptionInput(const TekGuiOption* option) {
386     // i feel like we should assume that bogus option type is
not an input... oh well
387     // these 3 are not writeable
388     switch (option->type) {
389     case TEK_LABEL:
390     case TEK_BUTTON_INPUT:
391     case TEK_UNKNOWN_INPUT:
392         return 0;
393     default:
394         return 1; // therefore everything else must be lol
395     }
396 }
397
398 /**
399 * Update a single input option in the array of options
400 * @param window The window containing all the options

```

```

401 * @param key The name of the option to be updated.
402 * @throws HASHTABLE_EXCEPTION if the option does not exist
403 */
404 static exception tekGuiUpdateInputOption(TekGuiOptionWindow*
window, const char* key) {
405     for (uint i = 0; i < window->len_options; i++) {
406         // get ith item of array
407         TekGuiOption* option = window->option_display + i;
408         // if not an option, cannot be this one
409         if (!tekGuiIsOptionInput(option))
410             continue;
411         // if name of option matches key, attempt to update
the option text
412         if (!strcmp(option->display.input.name, key)) {
413             tekChainThrow(tekGuiUpdateOptionInputText(window,
option));
414         }
415     }
416     return SUCCESS;
417 }
418
419 /**
420 * Write a string into a named option.
421 * @param window The window to write the string to.
422 * @param key The name of the option to write the string into
423 * @param string The string to be written
424 * @param len_string The length of the string to be written
425 * @throws MEMORY_EXCEPTION if malloc() fails.
426 * @throws HASHTABLE_EXCEPTION if the key does not exist.
427 */
428 exception tekGuiWriteStringOption(TekGuiOptionWindow* window,
const char* key, const char* string, const uint len_string) {
429     // get option data by key
430     TekGuiOptionData* option_data;
431     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
432
433     // free string if it already exists
434     if (option_data->data.string)
435         free(option_data->data.string);
436
437     // allocate new memory for string

```

```

438     option_data->data.string = (char*)malloc(len_string * sizeof(char));
439     if (!option_data->data.string)
440         tekThrow(MEMORY_EXCEPTION, "Failed to allocate string buffer.");
441
442     // copy string into new buffer
443     memcpy(option_data->data.string, string, len_string);
444
445     // visual update
446     tekChainThrow(tekGuiUpdateInputOption(window, key));
447     return SUCCESS;
448 }
449
450 /**
451 * Write a number to a named input option.
452 * @param window The window to write to.
453 * @param key The name of the option to write to.
454 * @param number The number to be written.
455 * @throws HASHTABLE_EXCEPTION if the key does not exist.
456 */
457 exception tekGuiWriteNumberOption(TekGuiOptionWindow* window,
const char* key, const double number) {
458     // get option from hashtable and copy number into it
459     TekGuiOptionData* option_data;
460     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
461     option_data->data.number = number;
462     tekChainThrow(tekGuiUpdateInputOption(window, key));
463     return SUCCESS;
464 }
465
466 /**
467 * Write a boolean value to a named input option.
468 * @param window The window to write to.
469 * @param key The name of the option to write to.
470 * @param boolean The boolean value to be written.
471 * @throws HASHTABLE_EXCEPTION if the key does not exist.
472 */
473 exception tekGuiWriteBooleanOption(TekGuiOptionWindow* window,
const char* key, const flag boolean) {
474     // get option from hashtable and copy boolean into it
475     TekGuiOptionData* option_data;

```

```

476     tekChainThrow(hashtableGet(&window->option_data, key,
477     (void**)&option_data));
478     option_data->data.boolean = boolean;
479     tekChainThrow(tekGuiUpdateInputOption(window, key));
480     return SUCCESS;
481 }
482 /**
483 * Write a vec3 to a named input option.
484 * @param window The window to write to.
485 * @param key The name of the option to write to.
486 * @param vector The vec3 to be written.
487 * @throws HASHTABLE_EXCEPTION if the key does not exist.
488 */
489 exception tekGuiWriteVec3Option(TekGuiOptionWindow* window,
const char* key, vec3 vector) {
490     // get option from hashtable and copy vec3 data into it
491     TekGuiOptionData* option_data;
492     tekChainThrow(hashtableGet(&window->option_data, key,
493     (void**)&option_data));
494     glm_vec3_copy(vector, option_data->data.vector_3);
495     tekChainThrow(tekGuiUpdateInputOption(window, key));
496     return SUCCESS;
497 }
498 /**
499 * Write a vec4 to a named input option.
500 * @param window The window to write to.
501 * @param key The name of the option to write to.
502 * @param vector The vec4 to be written.
503 * @throws HASHTABLE_EXCEPTION if the key does not exist.
504 */
505 exception tekGuiWriteVec4Option(TekGuiOptionWindow* window,
const char* key, vec4 vector) {
506     // get option from hashtable and copy vec4 data into it
507     TekGuiOptionData* option_data;
508     tekChainThrow(hashtableGet(&window->option_data, key,
509     (void**)&option_data));
510     glm_vec4_copy(vector, option_data->data.vector_4);
511     tekChainThrow(tekGuiUpdateInputOption(window, key));
512     return SUCCESS;
513 }
```

```

514 /**
515  * Write a number option given by a decimal numeric string.
516 Will convert the string to a number, or will be 0 if the string
517  * is not a valid number.
518  * @param window The window to write the number to
519  * @param key The name of the option to write the number to
520  * @param number_str The string containing the numeric data.
521  * @throws HASHTABLE_EXCEPTION if the key does not exist.
522 */
523 static exception
tekGuiWriteNumberOptionString(TekGuiOptionWindow* window, const
char* key, const char* number_str) {
524     char* endptr;
525     const uint len_number_str = strlen(number_str) + 1;
526     double number = strtod(number_str, &endptr);
527
528     if (errno == ERANGE)
529         number = 0.0;
530     if (endptr != number_str + len_number_str - 1) // if didnt
make it thru the string and terminated early = badness
531         number = 0.0;
532
533     // update with converted val
534     tekChainThrow(tekGuiWriteNumberOption(window, key,
number));
535     tekChainThrow(tekGuiUpdateInputOption(window, key));
536     return SUCCESS;
537
538 /**
539  * Write a boolean option given by a string. Will convert the
string to a boolean, or will be false if the string
540  * is not a valid boolean. Will ignore case and accept either
"True", "Yes", or "OK" as being true, anything else is false.
541  * @param window The window to write the boolean to
542  * @param key The name of the option to write the boolean to
543  * @param boolean_str The string containing the boolean data.
544  * @throws HASHTABLE_EXCEPTION if the key does not exist.
545 */
546 static exception
tekGuiWriteBooleanOptionString(TekGuiOptionWindow* window, const
char* key, const char* boolean_str) {
547     flag boolean = 0;

```

```

548      // accepted as true = true, yes or ok. no scientific
evidence that thats the best but ok
549      // anything else = false
550      if (!strcasecmp(boolean_str, "TRUE") || !
strcasecmp(boolean_str, "YES") || !strcasecmp(boolean_str, "OK"))
551          boolean = 1;
552
553      // update once checked
554      tekChainThrow(tekGuiWriteBooleanOption(window, key,
boolean));
555      tekChainThrow(tekGuiUpdateInputOption(window, key));
556      return SUCCESS;
557 }
558
559 /**
560 * Write an element of a vector given by a decimal numeric
string. Will convert the string to a number, or will be 0 if the
string
561 * is not a valid number. Will write to the index of the
vector specified.
562 * @param window The window to write the element to
563 * @param key The name of the option to write the element to
564 * @param element_str The string containing the numeric data.
565 * @param index The index of the vector to write at.
566 * @throws HASHTABLE_EXCEPTION if the key does not exist.
567 */
568 static exception
tekGuiWriteVecIndexOptionString(TekGuiOptionWindow* window, const
char* key, const char* element_str, const uint index) {
569     TekGuiOptionData* option_data;
570     tekChainThrow(hashtableGet(&window->option_data, key,
(void**)&option_data));
571
572     // cast this element string to a float
573     char* endptr;
574     uint len_element_str = strlen(element_str) + 1;
575     float element = strtod(element_str, &endptr);
576     if (errno == ERANGE)
577         element = 0.0f;
578     if (endptr != element_str + len_element_str - 1)
579         element = 0.0f;
580

```

```

581     // vector_3 and vector_4 overlap, so doesn't matter if
this was called for a vec3 or vec4.
582     option_data->data.vector_4[index] = element;
583     tekChainThrow(tekGuiUpdateInputOption(window, key));
584     return SUCCESS;
585 }
586
587 /**
588  * Writes a default value into an input option. For example, a
string -> nullptr, number -> 0.0, boolean -> false.
589  * @param window The window to write the value into.
590  * @param name The name of the option to write into.
591  * @param type The type of the option to write as.
592  * @throws FAILURE if type is not allowed.
593  * @throws HASHTABLE_EXCEPTION if name is not an option.
594 */
595 static exception tekGuiWriteDefaultValue(TekGuiOptionWindow*
window, const char* name, const flag type) {
596     // mallocate the option data
597     TekGuiOptionData* option_data =
(TekGuiOptionData*)malloc(sizeof(TekGuiOptionData));
598     if (!option_data)
599         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for option data.");
600     option_data->type = type;
601
602     // set the data based on type.
603     // default value is 0 or nullptr for a string
604     switch (type) {
605     case TEK_STRING_INPUT:
606         option_data->data.string = 0;
607         break;
608     case TEK_NUMBER_INPUT:
609         option_data->data.number = 0.0;
610         break;
611     case TEK_BOOLEAN_INPUT:
612         option_data->data.boolean = 0;
613         break;
614     case TEK_VEC3_INPUT:
615         glm_vec3_zero(option_data->data.vector_3);
616         break;
617     case TEK_VEC4_INPUT:
618         glm_vec4_zero(option_data->data.vector_4);

```

```

619         break;
620     default:
621         free(option_data);
622         tekThrow(FAILURE, "Unknown option type.");
623     }
624
625     // set the pointer at hash table.
626     tekChainThrow(hashtableSet(&window->option_data, name,
option_data));
627     return SUCCESS;
628 }
629
630 /**
631 * Callback for when text is inputted into one of the options.
The edited text input is specified as the first parameter.
632 * Also provides a pointer to the new text and its length.
Gives a pointer to the actual text, so you shouldn't overwrite
633 * it.
634 * @param text_input A pointer to the text input that called
back.
635 * @param text The new text that was just inputted.
636 * @param len_text The length of the new text.
637 * @throws EXCEPTION whatever the callback function throws
638 */
639 static exception tekGuiOptionInputCallback(TekGuiTextInput*
text_input, const char* text, const uint len_text) {
640     // grab necessary data
641     struct TekGuiOptionPair* option_pair = (struct
TekGuiOptionPair*)text_input->data;
642     TekGuiOptionWindow* window = option_pair->window;
643     TekGuiOption* option = option_pair->option;
644     const char* name = option->display.input.name;
645     const uint index = option->display.input.index;
646
647     switch (option->type) { // select correct method based on
type of option
648     case TEK_STRING_INPUT:
649         tekChainThrow(tekGuiWriteStringOption(window, name,
text, len_text));
650         break;
651     case TEK_NUMBER_INPUT:
652         tekChainThrow(tekGuiWriteNumberOptionString(window,
name, text));

```

```

653         break;
654     case TEK_BOOLEAN_INPUT:
655         tekChainThrow(tekGuiWriteBooleanOptionString(window,
name, text));
656         break;
657     case TEK_VEC3_INPUT:
658     case TEK_VEC4_INPUT:
659         tekChainThrow(tekGuiWriteVecIndexOptionString(window,
name, text, index));
660         break;
661     default:
662         tekThrow(FAILURE, "Input called on unknown input
type.");
663     }
664
665     // call the callback
666     TekGuiOptionWindowCallbackData callback_data;
667     callback_data.type = option->type;
668     callback_data.name = name;
669     if (window->callback) tekChainThrow(window-
>callback(window, callback_data));
670
671     // update text in case the text was changed during the
callback method
672     tekChainThrow(tekGuiUpdateOptionInputText(window,
option));
673
674     return SUCCESS;
675 }
676
677 /**
678 * Callback for when a button option is clicked. All buttons
will link to this callback, but callback receives a pointer
679 * to the button which called it. Will simply pass the
callback on to the handler specified by the user.
680 * @param button A pointer to the button which called back
681 * @param callback_data Callback data, which button was
pressed / released
682 */
683 static void tekGuiButtonInputCallback(TekGuiTextButton*
button, TekGuiButtonCallbackData callback_data) {
684     struct TekGuiOptionPair* option_pair = (struct
TekGuiOptionPair*)button->data;

```

```

685     TekGuiOptionWindow* window = option_pair->window;
686     const TekGuiOption* option = option_pair->option;
687
688     // if there is no callback just instantly quit
689     if (!window->callback)
690         return;
691
692     // make sure that there is a left click or else quit
693     if (callback_data.type !=
TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK
694         || callback_data.data.mouse_button.action !=
GLFW_RELEASE
695         || callback_data.data.mouse_button.button !=
GLFW_MOUSE_BUTTON_LEFT)
696         return;
697
698     // fill up some callback data
699     TekGuiOptionWindowCallbackData outbound_callback_data;
700     outbound_callback_data.type = TEK_BUTTON_INPUT;
701     outbound_callback_data.name = option->display.button.name;
702
703     // pass control onto the user
704     window->callback(window, outbound_callback_data);
705 }
706
707 /**
708 * Create a label for the window and add it to the list of
displayed items.
709 * @param[in] label The text that the label should display.
710 * @param[in] text_height The height of the text for this
label.
711 * @param[in/out] option_display The array for which to add
the label display to.
712 * @param[in/out] option_index A pointer to the last written
index of the array, will be incremented as more displays are added.
713 * @throws OPENGL_EXCEPTION if something fails graphically.
714 */
715 static exception tekGuiCreateLabelOption(const char* label,
const uint text_height, TekGuiOption* option_display, uint*
option_index) {
716     // empty option struct
717     TekGuiOption* option = option_display + *option_index;
718     (*option_index)++;

```

```

719
720     // fill with label data
721     option->type = TEK_LABEL;
722     option->height = text_height * 5 / 4;
723     TekBitmapFont* font;
724     tekChainThrow(tekGuiGetDefaultFont(&font));
725     tekChainThrow(tekCreateText(label, text_height, font,
&option->display.label));
726
727     return SUCCESS;
728 }
729
730 /**
731  * Create a single input display and add it to the list of
displayed items.
732  * @param[in] window The window which the input belongs to.
733  * @param[in] name The name of the input, will be used to
access it.
734  * @param[in] type The type of the input, something like
TEK_..._INPUT.
735  * @param[in] input_width The width of each input to create.
736  * @param[in/out] option_display The array for which to add
the option display to.
737  * @param[in/out] option_index A pointer to the last written
index of the array, will be incremented as more displays are added.
738  * @throws OPENGL_EXCEPTION if something graphical fails.
739 */
740 static exception tekGuiCreateSingleInput(TekGuiOptionWindow*
window, const char* name, const flag type, const uint input_width,
TekGuiOption* option_display, uint* option_index) {
741     // get pointer to struct
742     TekGuiOption* option = option_display + *option_index;
743     (*option_index)++;
744
745     // fill with input data
746     option->type = type;
747     tekChainThrow(tekGuiCreateTextInput(&option-
>display.input.text_input));
748     option->height = option-
>display.input.text_input.button.hitbox_height;
749     tekChainThrow(tekGuiSetTextInputSize(&option-
>display.input.text_input, input_width, option->height));
750

```

```

751     // set up callbacks
752     struct TekGuiOptionPair* option_pair = (struct
TekGuiOptionPair*)malloc(sizeof(struct TekGuiOptionPair));
753     if (!option_pair)
754         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for option pair.");
755     option_pair->window = window;
756     option_pair->option = option;
757     option->display.input.text_input.data = option_pair;
758     option->display.input.text_input.callback =
tekGuiOptionInputCallback;
759     option->display.input.name = name;
760     option->display.input.index = 0; // index only relevant
for multi-item inputs
761
762     // add an empty entry in the hash table.
763     tekChainThrow(tekGuiWriteDefaultValue(window, name,
type));
764     tekChainThrow(tekGuiUpdateOptionInputText(window,
option_display + *option_index - 1))
765
766     return SUCCESS;
767 }
768
769 /**
770  * Create multiple input displays which are all linked to the
same name, used for vector inputs
771  * @param[in] window The window which the input belongs to.
772  * @param[in] name The name of the input, used to access the
inputted data.
773  * @param[in] type The input type, something like
TEK_..._INPUT.
774  * @param[in] input_width The width of each input to create.
775  * @param[in] num_inputs The number of inputs to add.
776  * @param[in/out] option_display The array for which to add
the option displays to.
777  * @param[in/out] option_index A pointer to the last written
index of the array, will be incremented as more displays are added.
778  * @throws OPENGL_EXCEPTION if any of the single inputs failed
to be created.
779 */
780 static exception tekGuiCreateMultiInput(TekGuiOptionWindow*
window, const char* name, const flag type, const uint input_width,

```

```

const uint num_inputs, TekGuiOption* option_display, uint*
option_index) {
    781     // iterate for each input needed
    782     for (uint i = 0; i < num_inputs; i++) {
    783         // get the option pointer before the index is
incremented.
    784         TekGuiOption* option = option_display + *option_index;
    785
    786         // each has the same name, as pointing to the same
option
    787         tekChainThrow(tekGuiCreateSingleInput(window, name,
type, input_width, option_display, option_index));
    788
    789         // once created, update the display index to whatever
    790         option->display.input.index = i;
    791     }
    792
    793     return SUCCESS;
    794 }
    795
    796 /**
    797     * Create a button type option and add it to the window. Will
add to the list of options, and set up callbacks.
    798     * @param window The window to add the option to.
    799     * @param name The name of the new button to be created.
    800     * @param label The text to be displayed on the button when it
is created
    801     * @param option_display The array of option displays
    802     * @param option_index The next free index in the option
array.
    803     * @throws MEMORY_EXCEPTION if malloc() fails.
    804 */
    805 static exception tekGuiCreateButtonOption(TekGuiOptionWindow*
window, const char* name, const char* label, TekGuiOption*
option_display, uint* option_index) {
    806     // empty option struct
    807     TekGuiOption* option = option_display + *option_index;
    808     (*option_index)++;
    809
    810     // set all the option data
    811     option->type = TEK_BUTTON_INPUT;
    812     tekChainThrow(tekGuiCreateTextButton(label, &option-
>display.button.button));

```

```

813     option->height = option-
>display.button.button.button.hitbox_height + 10; // fudge factor
🔥🔥
814     option->display.button.name = name;
815
816     // set up callbacks
817     // option pair = a collection of data that is needed in
the callback (the window and the button option)
818     struct TekGuiOptionPair* option_pair = (struct
TekGuiOptionPair*)malloc(sizeof(struct TekGuiOptionPair));
819     if (!option_pair)
820         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for option pair.");
821     option_pair->window = window;
822     option_pair->option = option;
823     option->display.button.button.callback =
tekGuiButtonInputCallback;
824     option->display.button.button.data = (void*)option_pair;
825
826     return SUCCESS;
827 }
828
829 /**
830 * Add a chunk of option displays based on a set of
parameters. For example, a vec3 option will create 4 option
displays, one for the label, and 3 inputs.
831 * @param[in] window The window which to create the options
for.
832 * @param name The name of the option display, used to
retrieve the data later on.
833 * @param label The label, what will be displayed on the gui.
834 * @param type The type of option, in the form TEK_..._INPUT
or TEK_LABEL
835 * @param text_height The height of the label text.
836 * @param input_width The width of the text inputs.
837 * @param option_display An array of option displays to add
to.
838 * @param option_index The last written index of the
option_display array.
839 * @throws OPENGL_EXCEPTION if something went wrong
graphically at any stage.
840 */

```

```

841 static exception tekGuiCreateOption(TekGuiOptionWindow*
window, const char* name, const char* label, const flag type, const
uint text_height, const uint input_width, TekGuiOption*
option_display, uint* option_index) {
842     // a button does not require a label, cuz it already has
text on it
843     if (type == TEK_BUTTON_INPUT) {
844         tekChainThrow(tekGuiCreateButtonOption(window, name,
label, option_display, option_index));
845         return SUCCESS;
846     }
847
848     // for non-buttons, write a piece of text above
849     tekChainThrow(tekGuiCreateLabelOption(label, text_height,
option_display, option_index));
850
851     // then, append the correct type of input below.
852     switch (type) {
853     case TEK_LABEL:
854         return SUCCESS; // no inputs
855     case TEK_STRING_INPUT:
856     case TEK_NUMBER_INPUT:
857     case TEK_BOOLEAN_INPUT: // 1 input
858         tekChainThrow(tekGuiCreateSingleInput(window, name,
type, input_width, option_display, option_index));
859         break;
860     case TEK_VEC3_INPUT: // 3 inputs
861         tekChainThrow(tekGuiCreateMultiInput(window, name,
type, input_width, 3, option_display, option_index));
862         break;
863     case TEK_VEC4_INPUT: // 4 inputs
864         tekChainThrow(tekGuiCreateMultiInput(window, name,
type, input_width, 4, option_display, option_index));
865         break;
866     default:
867         tekThrow(YML_EXCEPTION, "Invalid input type for
option.");
868     }
869
870     return SUCCESS;
871 }
872
873 /**

```

```

874     * Draw a label, just a piece of text.
875     * @param option A pointer to the option display where the
label is stored.
876     * @param x_pos The x coordinate to draw at.
877     * @param y_pos The y coordinate to draw at.
878     * @throws OPENGL_EXCEPTION if something fails graphically.
879     */
880 static exception tekGuiDrawOptionLabel(const TekGuiOption*
option, const int x_pos, const int y_pos) {
881     // figure out correct position and draw
882     float x = (float)x_pos + (float)option->height * 0.4f; // slight shift to the right so first letter isnt touching the edge
883     float y = (float)y_pos;
884     tekChainThrow(tekDrawText(&option->display.label, x, y));
885     return SUCCESS;
886 }
887
888 /**
889     * Draw an option input, which is a text input + link back to the option table.
890     * @param option A pointer to the option display to draw.
891     * @param x_pos The x coordinate to draw at.
892     * @param y_pos The y coordinate to draw at.
893     * @throws OPENGL_EXCEPTION if something fails graphically.
894     */
895 static exception tekGuiDrawOptionInput(TekGuiOption* option,
const int x_pos, const int y_pos) {
896     // move to position and draw
897     tekChainThrow(tekGuiSetTextInputPosition(&option-
>display.input.text_input, x_pos + option->height, y_pos));
898     tekChainThrow(tekGuiDrawTextInput(&option-
>display.input.text_input));
899     return SUCCESS;
900 }
901
902 /**
903     * Draw a button input, which is just a text button + link back to main window.
904     * @param option The option to draw.
905     * @param x_pos The x coordinate to draw at.
906     * @param y_pos The y coordinate to draw at.
907     * @throws OPENGL_EXCEPTION .
908     */

```

```

909 static exception tekGuiDrawButtonInput(TekGuiOption* option,
const int x_pos, const int y_pos) {
910     // move to correct position and draw.
911     tekChainThrow(tekGuiSetTextButtonPosition(&option-
>display.button.button, x_pos + option->height * 0.4f, y_pos + 5));
912     tekChainThrow(tekGuiDrawTextButton(&option-
>display.button.button));
913     return SUCCESS;
914 }
915
916 /**
917  * Draw an option display, this call will decide the
appropriate method of drawing based on the type.
918  * @param option A pointer to the option display to draw.
919  * @param x_pos The x coordinate to draw at.
920  * @param y_pos The y coordinate to draw at.
921  * @throws OPENGL_EXCEPTION if something goes wrong
graphically.
922 */
923 static exception tekGuiDrawOption(TekGuiOption* option, const
int x_pos, const int y_pos) {
924     switch (option->type) { // use the corresponding method
for the type of option being drawn.
925         case TEK_LABEL:
926             tekGuiDrawOptionLabel(option, x_pos, y_pos); // draw
only text
927             break;
928         case TEK_BUTTON_INPUT:
929             tekGuiDrawButtonInput(option, x_pos, y_pos); // draw
button
930             break;
931         case TEK_STRING_INPUT:
932         case TEK_NUMBER_INPUT:
933         case TEK_BOOLEAN_INPUT:
934         case TEK_VEC3_INPUT:
935         case TEK_VEC4_INPUT:
936             tekGuiDrawOptionInput(option, x_pos, y_pos); // further
differentiate, label + some amount of text inputs
937             break;
938         default:
939             tekThrow(FAILURE, "Cannot draw option of unknown
type.");
940     }

```

```

941
942     return SUCCESS;
943 }
944
945 /**
946  * Bring the internal button collider of an option to the
947 front of the button list, so that it will receive mouse inputs
948  * before other buttons on the screen, and appear to be above
949 them.
950  */
951 static exception tekGuiBringOptionToFront(TekGuiOption*
option) {
952     switch (option->type) {
953         // an actual button input should have its button moved to
the front
954         case TEK_BUTTON_INPUT:
955             tekChainThrow(tekGuiBringButtonToFront(&option-
>display.button.button.button));
956             break;
957         // text-based inputs should have their "select this input"
button moved.
958         case TEK_STRING_INPUT:
959         case TEK_NUMBER_INPUT:
960         case TEK_BOOLEAN_INPUT:
961         case TEK_VEC3_INPUT:
962         case TEK_VEC4_INPUT:
963             tekChainThrow(tekGuiBringButtonToFront(&option-
>display.input.text_input.button));
964             break;
965         default:
966             break;
967     }
968     return SUCCESS;
969 }
970
971 /**
972  * Draw callback function for option window, called after the
base window is drawn.
973  * @param window_ptr The base window.
974  * @throws OPENGL_EXCEPTION if something fails graphically.
975  */

```

```

976 static exception tekGuiOptionWindowDrawCallback(TekGuiWindow*
window_ptr) {
977     // get the option window
978     TekGuiOptionWindow* window =
(TekGuiOptionWindow*)window_ptr->data;
979
980     // for each option, draw it above the base window
981     int x_pos = window_ptr->x_pos;
982     int y_pos = window_ptr->y_pos;
983     for (uint i = 0; i < window->len_options; i++) {
984         TekGuiOption* option = window->option_display + i;
985         tekChainThrow(tekGuiDrawOption(option, x_pos, y_pos));
986         y_pos += (int)option->height; // increment height each
time to avoid drawing over last option
987     }
988
989     return SUCCESS;
990 }
991
992 /**
993 * Called every time the base window is brought to the front.
994 * @param window_ptr Pointer to the base window.
995 * @throws LIST_EXCEPTION .
996 */
997 static exception
tekGuiOptionWindowSelectCallback(TekGuiWindow* window_ptr) {
998     // get the option window
999     TekGuiOptionWindow* window =
(TekGuiOptionWindow*)window_ptr->data;
1000
1001     // for each option, bring it to the front
1002     for (uint i = 0; i < window->len_options; i++) {
1003         TekGuiOption* option = window->option_display + i;
1004         tekChainThrow(tekGuiBringOptionToFront(option));
1005     }
1006
1007     return SUCCESS;
1008 }
1009
1010 /**
1011 * Create the base window of the option window. Simple stuff
like setting size, position and title.
1012 * @param window The option window to create the window for.

```

```

1013     * @param defaults The default settings to apply to the
window.
1014     * @throws MEMORY_EXCEPTION for a lot of the text buffer
related things that can fail.
1015 */
1016 static exception tekGuiCreateBaseWindow(TekGuiOptionWindow*
window, struct TekGuiOptionsWindowDefaults* defaults) {
1017     // boring stuff for the window
1018     tekChainThrow(tekGuiCreateWindow(&window->window));
1019     tekGuiSetWindowPosition(&window->window, defaults->x_pos,
defaults->y_pos);
1020     tekGuiSetWindowSize(&window->window, defaults->width,
defaults->height);
1021     tekChainThrow(tekGuiSetWindowTitle(&window->window,
defaults->title));
1022     window->window.draw_callback =
tekGuiOptionWindowDrawCallback; // called every time drawn
1023     window->window.select_callback =
tekGuiOptionWindowSelectCallback; // called every time sent to front
1024     window->window.data = window;
1025     return SUCCESS;
1026 }
1027
1028 /**
1029     * Get the size (the number of option displays required) to
fit a type of option display.
1030     * @param type The type of option display.
1031     * @return The size.
1032 */
1033 static uint tekGuiGetOptionDisplaySize(const flag type) {
1034     switch (type) {
1035     case TEK_LABEL:
1036     case TEK_BUTTON_INPUT: // just the button
1037         return 1;
1038     case TEK_STRING_INPUT:
1039     case TEK_NUMBER_INPUT:
1040     case TEK_BOOLEAN_INPUT: // label + input
1041         return 2;
1042     case TEK_VEC3_INPUT: // label + 3 inputs
1043         return 4;
1044     case TEK_VEC4_INPUT: // label + 4 inputs
1045         return 5;
1046     default: // god nose

```

```

1047         return 0;
1048     }
1049 }
1050
1051 /**
1052 * Get the total size of an array of options, for example a
1053 * label, a vec3 input and a string input:
1054 * @code
1055 * 1 A Label
1056 * 2 A Vec3 Input:
1057 * 3 [    ]
1058 * 4 [    ]
1059 * 5 [    ]
1060 * 6 A String Input:
1061 * 7 [    ]
1062 * @endcode
1063 * Will be a size of 7.
1064 * @param options The array of options.
1065 * @param len_options The number of options in the array.
1066 * @return The total size.
1067 */
1068 static uint tekGuiGetOptionsDisplayTotalSize(struct
TekGuiOptionsWindowOption* options, const uint len_options) {
1069     // sum length of each individual component
1070     uint total_len = 0;
1071     for (uint i = 0; i < len_options; i++) {
1072         total_len +=
1073             tekGuiGetOptionDisplaySize(options[i].type);
1074     }
1075
1076 /**
1077 * Create an option window from a yml file.
1078 * @param options_yml The filepath of the yml file.
1079 * @param window A pointer to an empty struct which will be
1080 * filled with the option window data.
1081 * @throws OPENGL_EXCEPTION if something fails graphically.
1082 * @throws MEMORY_EXCEPTION in a few scenarios to do with
1083 * allocating buffers for text/titles.
1084 */
1085 exception tekGuiCreateOptionWindow(const char* options_yml,
TekGuiOptionWindow* window) {

```

```

1084     // options stored in hashtable accessed by name
1085     tekChainThrow(hashtableCreate(&window->option_data, 4));
1086
1087     // read yml file to specify how the window should be layed
out
1088     YmlFile yml_file = {};
1089     tekChainThrow(ymlReadFile(options_yml, &yml_file));
1090
1091     // read default options and yml layout for rendering
1092     struct TekGuiOptionsWindowDefaults defaults = {};
1093     struct TekGuiOptionsWindowOption* options;
1094     uint len_options = 0;
1095     tekChainThrowThen(tekGuiLoadOptionsYml(&yml_file,
&defaults, &options, &len_options), {
1096         ymlDelete(&yml_file);
1097     });
1098
1099     // allocate memory for options
1100     window->len_options =
tekGuiGetOptionsDisplayTotalSize(options, len_options);
1101     window->option_display = (TekGuiOption*)malloc(window-
>len_options * sizeof(TekGuiOption));
1102     if (!window->option_display)
1103         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for options display array.");
1104
1105     // create gui elements for all the options
1106     uint option_index = 0;
1107     for (uint i = 0; i < len_options; i++) {
1108         tekChainThrowThen(tekGuiCreateOption(window,
options[i].name, options[i].label, options[i].type,
defaults.text_height, defaults.input_width, window->option_display,
&option_index), {
1109             ymlDelete(&yml_file);
1110         });
1111     }
1112
1113     // create the actual window
1114     tekChainThrowThen(tekGuiCreateBaseWindow(window,
&defaults), {
1115         ymlDelete(&yml_file);
1116     });
1117

```

```

1118     // yml file no longer needed, get rid of it
1119     ymlDelete(&yml_file);
1120     return SUCCESS;
1121 }
1122
1123 /**
1124 * Free any memory allocated by a displayed option.
1125 * @param option The option to delete.
1126 */
1127 static void tekGuiDeleteOption(TekGuiOption* option) {
1128     switch (option->type) { // use method corresponding to the
option type
1129     case TEK_LABEL:
1130         tekDeleteText(&option->display.label);
1131         break;
1132     case TEK_BUTTON_INPUT:
1133         tekGuiDeleteTextButton(&option-
>display.button.button);
1134         free((void*)option->display.button.name); // free name
1135         break;
1136     case TEK_STRING_INPUT: // all use the same method to
delete, as do not require further memory
1137     case TEK_NUMBER_INPUT:
1138     case TEK_BOOLEAN_INPUT:
1139     case TEK_VEC3_INPUT:
1140     case TEK_VEC4_INPUT:
1141         tekGuiDeleteTextInput(&option-
>display.input.text_input);
1142         if (option->display.input.index == 0) // multi-options
share the same name, only free it once.
1143             free((void*)option->display.input.name); // free
name
1144             break;
1145     default:
1146         break;
1147     }
1148 }
1149
1150 /**
1151 * Free any memory allocated by the option window.
1152 * @param window The window to delete.
1153 */
1154 void tekGuiDeleteOptionWindow(TekGuiOptionWindow* window) {

```

```

1155     // delete the options
1156     for (uint i = 0; i < window->len_options; i++) {
1157         TekGuiOption* option = window->option_display + i;
1158         tekGuiDeleteOption(option);
1159     }
1160
1161     // delete the option data.
1162     // first get all the option data from hashtable
1163     TekGuiOptionData* option_data_array;
1164     hashtableGetValues(&window->option_data,
&option_data_array);
1165     if (!option_data_array) return;
1166
1167     // final cleanup
1168     free(option_data_array);
1169     hashtableDelete(&window->option_data);
1170     tekGuiDeleteWindow(&window->window);
1171 }
```

tekgui/option_window.h

```

1 #pragma once
2
3 #include "text_button.h"
4 #include "window.h"
5 #include "../core/hashtable.h"
6 #include "../core/vector.h"
7 #include "../tekgl/text.h"
8 #include "text_input.h"
9
10 #define TEK_UNKNOWN_INPUT (-1)
11 #define TEK_LABEL 0
12 #define TEK_STRING_INPUT 1
13 #define TEK_NUMBER_INPUT 2
14 #define TEK_BOOLEAN_INPUT 3
15 #define TEK_VEC3_INPUT 4
16 #define TEK_VEC4_INPUT 5
17 #define TEK_BUTTON_INPUT 6
18
19 typedef struct TekGuiOptionData {
20     flag type;
21     union {
22         char* string;
23         double number;
```

```

24         flag boolean;
25         vec3 vector_3;
26         vec4 vector_4;
27     } data;
28 } TekGuiOptionData;
29
30 typedef struct TekGuiOption {
31     flag type;
32     uint height;
33     union {
34         TekText label;
35         struct {
36             TekGuiTextInput text_input;
37             const char* name;
38             uint index;
39         } input;
40         struct {
41             TekGuiTextButton button;
42             const char* name;
43             } button;
44     } display;
45 } TekGuiOption;
46
47 typedef struct TekGuiOptionWindowCallbackData {
48     flag type;
49     const char* name;
50 } TekGuiOptionWindowCallbackData;
51
52 struct TekGuiOptionWindow;
53 typedef exception (*TekGuiOptionWindowCallback)(struct
TekGuiOptionWindow* window, TekGuiOptionWindowCallbackData
callback_data);
54
55 typedef struct TekGuiOptionWindow {
56     TekGuiWindow window;
57     TekGuiOption* option_display;
58     uint len_options;
59     HashTable option_data;
60     void* data;
61     TekGuiOptionWindowCallback callback;
62 } TekGuiOptionWindow;
63

```

```

64 exception tekGuiCreateOptionWindow(const char* options_yml,
TekGuiOptionWindow* window);
65 exception tekGuiWriteStringOption(TekGuiOptionWindow* window,
const char* key, const char* string, uint len_string);
66 exception tekGuiWriteNumberOption(TekGuiOptionWindow* window,
const char* key, double number);
67 exception tekGuiWriteBooleanOption(TekGuiOptionWindow* window,
const char* key, flag boolean);
68 exception tekGuiWriteVec3Option(TekGuiOptionWindow* window,
const char* key, vec3 vector);
69 exception tekGuiWriteVec4Option(TekGuiOptionWindow* window,
const char* key, vec4 vector);
70 exception tekGuiReadStringOption(TekGuiOptionWindow* window,
const char* key, char** string);
71 exception tekGuiReadNumberOption(TekGuiOptionWindow* window,
const char* key, double* number);
72 exception tekGuiReadBooleanOption(TekGuiOptionWindow* window,
const char* key, flag* boolean);
73 exception tekGuiReadVec3Option(TekGuiOptionWindow* window, const
char* key, vec3 vector);
74 exception tekGuiReadVec4Option(TekGuiOptionWindow* window, const
char* key, vec4 vector);
75 void tekGuiDeleteOptionWindow(TekGuiOptionWindow* window);

```

tekgui/primitives.c

```

1 #include "primitives.h"
2
3 #include <cglm/cam.h>
4 #include <cglm/mat4.h>
5 #include <cglm/vec2.h>
6
7 #include "../tekgl/entity.h"
8 #include "../tekgl/manager.h"
9 #include "../tekgl/shader.h"
10 #include "glad/glad.h"
11 #include "GLFW/glfw3.h"
12
13 mat4 projection;
14 uint line_shader_program = 0;
15 uint oval_shader_program = 0;
16 uint image_shader_program = 0;
17
18 /**

```

```

19   * Framebuffer resize callback for primitives code. Called
20 whenever window is resized.
21   * @param window_width The new width of the window.
22   * @param window_height The new height of the window.
23   */
24 void tekPrimitiveFramebufferCallback(const int window_width,
25 const int window_height) {
26     // recreate projection matrix for shaders.
27     glm_ortho(0.0f, (float)window_width, (float)window_height,
28               0.0f, -1.0f, 1.0f, projection);
29 }
30 /**
31  * Create a simple rectangular mesh from a top left and bottom
32 right position.
33  * @param point_a The top left coordinate of the rectangle.
34  * @param point_b The bottom right coordinate of the rectangle.
35  * @param mesh The outputted rectangle mesh.
36  * @throws OPENGL_EXCEPTION if could not create the mesh.
37  */
38 static exception tekGuiCreateRectangularMesh(vec2 point_a, vec2
point_b, TekMesh* mesh) {
39     // four vertices of the rectangle
40     const float vertices[] = {
41         point_a[0], point_a[1],
42         point_a[0], point_b[1],
43         point_b[0], point_b[1],
44         point_b[0], point_a[1]
45     };
46     // draw order, to split into two triangles.
47     const uint indices[] = {
48         0, 1, 2,
49         0, 2, 3
50     };
51     // layout 1x vec2 per vertex
52     const int layout[] = {
53         2
54     };
55     // create the mesh

```

```

56     tekChainThrow(tekCreateMesh(vertices, 8, indices, 6,
layout, 1, mesh));
57     return SUCCESS;
58 }
59
60 /**
61 * Create the mesh needed to draw an image - requires there to
be texture coordinates as well as positions.
62 * @param width The width of the image.
63 * @param height The height of the image.
64 * @param mesh The outputted mesh for the image.
65 * @throws MEMORY_EXCEPTION if malloc() fails.
66 */
67 static exception tekGuiCreateImageMesh(const float width, const
float height, TekMesh* mesh) {
68     // vertices with position + texture coordinate
69     const float vertices[] = {
70         0.0f, 0.0f, 0.0f, 1.0f,
71         0.0f, height, 0.0f, 0.0f,
72         width, height, 1.0f, 0.0f,
73         width, 0.0f, 1.0f, 1.0f
74     };
75
76     // indices / drawing order
77     const uint indices[] = {
78         0, 1, 2,
79         0, 2, 3
80     };
81
82     // layout - 2x vec2
83     const int layout[] = {
84         2, 2
85     };
86
87     // create the mesh
88     tekChainThrow(tekCreateMesh(vertices, 16, indices, 6,
layout, 2, mesh));
89     return SUCCESS;
90 }
91
92 /**
93 * Delete callback for primitives code, deletes shaders.
94 */

```

```

95 static void tekDeletePrimitives() {
96     // delete the shaders
97     tekDeleteShaderProgram(line_shader_program);
98     tekDeleteShaderProgram(oval_shader_program);
99     tekDeleteShaderProgram(image_shader_program);
100 }
101
102 /**
103 * The opengl load callback for the primitives code, sets up
the shaders needed to draw primitive shapes.
104 * @throws SHADER_EXCEPTION if could not create the shaders.
105 */
106 static exception tekGLLoadPrimitives() {
107     // create shaders
108
tekChainThrow(tekCreateShaderProgramVF("../shader/line_vertex.glvs",
"../shader/line_fragment.glfs", &line_shader_program));
109
tekChainThrow(tekCreateShaderProgramVF("../shader/oval_vertex.glvs",
"../shader/oval_fragment.glfs", &oval_shader_program));
110
tekChainThrow(tekCreateShaderProgramVF("../shader/image_vertex.glvs"
, "../shader/image_fragment.glfs", &image_shader_program));
111
112     // projection matrix stuff for the shaders
113     int window_width, window_height;
114
tekChainThrow(tekAddFramebufferCallback(tekPrimitiveFramebufferCallb
ack));
115     tekGetWindowSize(&window_width, &window_height);
116     glm_ortho(0.0f, (float)window_width, 0.0f,
(float)window_height, -1.0f, 1.0f, projection);
117     return SUCCESS;
118 }
119
120 /**
121 * Initialise the primitives code, setting up callbacks.
122 */
123 tek_init tekInitPrimitives() {
124     // set up opengl and delete callback.
125     tekAddGLLoadFunc(tekGLLoadPrimitives);
126     tekAddDeleteFunc(tekDeletePrimitives);
127 }
```

```

128
129  /**
130   * Create a line that connects two points with a certain
131   * thickness.
132   * @param point_a The first point of the line.
133   * @param point_b The second point of the line.
134   * @param thickness The thickness of the line in pixels.
135   * @param color The colour to fill the line with.
136   * @param line The outputted line.
137   */
138 exception tekGuiCreateLine(vec2 point_a, vec2 point_b, const
139 float thickness, vec4 color, TekGuiLine* line) {
140     // find the direction that the line is pointing, and
141     // normalise it
142     vec2 mod;
143     glm_vec2_sub(point_b, point_a, mod);
144     glm_vec2_normalize(mod);
145
146     // get the perpendicular vector to the direction
147     const float swap = mod[0];
148     mod[0] = mod[1];
149     mod[1] = -swap;
150
151     // make vector be same length as half the thickness
152     // adding on one side, subtracting on other side makes up
153     // whole thickness again
154     mod[0] *= thickness;
155     mod[1] *= thickness;
156
157     // create mesh data for the line
158     const float vertices[] = {
159         point_a[0] - mod[0], point_a[1] - mod[1],
160         point_a[0] + mod[0], point_a[1] + mod[1],
161         point_b[0] + mod[0], point_b[1] + mod[1],
162         point_b[0] - mod[0], point_b[1] - mod[1]
163     };
164
165     const uint indices[] = {
166         0, 1, 2,
167         0, 2, 3
168     };
169
170 }
```

```

167     const int layout[] = {
168         2
169     };
170
171     // create mesh and set color of line
172     tekChainThrow(tekCreateMesh(vertices, 8, indices, 6,
173     layout, 1, &line->mesh));
174     glm_vec4_copy(color, line->color);
175
176     return SUCCESS;
177 }
178 /**
179 * Draw a line to the screen.
180 * @param line The line to be drawn.
181 * @throws SHADER_EXCEPTION .
182 */
183 exception tekGuiDrawLine(const TekGuiLine* line) {
184     // use line shader
185     tekBindShaderProgram(line_shader_program);
186
187     // set uniforms
188     tekChainThrow(tekShaderUniformMat4(line_shader_program,
189     "projection", projection));
190     tekChainThrow(tekShaderUniformVec4(line_shader_program,
191     "line_color", line->color));
192
193     // draw
194     tekDrawMesh(&line->mesh);
195
196 /**
197 * Delete a line, freeing any associated memory.
198 * @param line The line to delete.
199 */
200 void tekGuiDeleteLine(const TekGuiLine* line) {
201     // delete mesh
202     tekDeleteMesh(&line->mesh);
203 }
204
205 /**

```

```

206     * Create an oval based on two points, the oval will occupy the
207     bounding box specified by the two points.
208
209     * @param point_a The top left point of the bounding box.
210     * @param point_b The bottom right point of the bounding box.
211     * @param thickness The thickness of the line of the oval.
212     * @param fill Should the oval be filled or not (1 or 0).
213     * @param color The colour to use for the oval.
214     * @param oval The outputted oval.
215     * @throws MEMORY_EXCEPTION if malloc() fails.
216     */
217
218     exception tekGuiCreateOval(vec2 point_a, vec2 point_b, const
219     float thickness, const flag fill, vec4 color, TekGuiOval* oval) {
220         // generic rectangle mesh
221         tekChainThrow(tekGuiCreateRectangularMesh(point_a, point_b,
222         &oval->mesh));
223
224         // some constants for this oval.
225         oval->center[0] = (point_a[0] + point_b[0]) * 0.5f;
226         oval->center[1] = (point_a[1] + point_b[1]) * 0.5f;
227
228         oval->inv_width = 2.0f / (point_b[0] - point_a[0]);
229         oval->inv_height = 2.0f / (point_b[1] - point_a[1]);
230
231         // fill works by saying, if distance to edge is greater
232         // than min_dist, colour in
233         // if filled, this should always be true
234         if (fill) {
235             oval->min_dist = 0.0f;
236             // otherwise, should only fill if near the edge
237             } else {
238                 const float radius = (point_b[0] - point_a[0]) * 0.5f;
239                 const float fill_dist = radius - thickness;
240                 oval->min_dist = oval->inv_width * fill_dist;
241             }
242
243         // set up colour
244         glm_vec4_copy(color, oval->color);
245
246         return SUCCESS;
247     }
248
249     /**
250      * Draw an oval to the screen.

```

```

245 * @param oval The oval to draw.
246 * @throws SHADER_EXCEPTION .
247 */
248 exception tekGuiDrawOval(const TekGuiOval* oval) {
249     // load shader
250     tekBindShaderProgram(oval_shader_program);
251
252     // set uniform
253     // works using an oval equation hence all this maths
254     tekChainThrow(tekShaderUniformMat4(oval_shader_program,
255 "projection", projection));
255     tekChainThrow(tekShaderUniformFloat(oval_shader_program,
256 "inv_width", oval->inv_width));
256     tekChainThrow(tekShaderUniformFloat(oval_shader_program,
257 "inv_height", oval->inv_height));
258     tekChainThrow(tekShaderUniformFloat(oval_shader_program,
259 "min_dist", oval->min_dist));
260     tekChainThrow(tekShaderUniformVec2(oval_shader_program,
261 "center", oval->center));
262     tekChainThrow(tekShaderUniformVec4(oval_shader_program,
263 "oval_color", oval->color));
264 }
265
266 /**
267 * Delete an oval, freeing any allocated memory. Archaic
268 function, was used in the noughts and crosses edition.
269 */
270 void tekGuiDeleteOval(const TekGuiOval* oval) {
271     // delete mesh
272     tekDeleteMesh(&oval->mesh);
273 }
274
275 /**
276 * Create an image from an image file, and create the mesh
277 needed to draw it.
278 * @param width The width of the image.
279 * @param height The height of the image.
280 * @param texture_filename The name of the texture file.

```

```

280     * @param image The outputted image.
281     * @throws STBI_EXCEPTION if could not load the image.
282     */
283 exception tekGuiCreateImage(const float width, const float
height, const char* texture_filename, TekGuiImage* image) {
284     // create the mesh and texture
285     tekChainThrow(tekGuiCreateImageMesh(width, height, &image-
>mesh));
286     tekChainThrowThen(tekCreateTexture(texture_filename,
&image->texture_id), {
287         tekDeleteMesh(&image->mesh);
288     });
289
290     return SUCCESS;
291 }
292
293 /**
294     * Draw an image to the screen at the specified position.
295     * @param image The image to be drawn.
296     * @param x The x-coordinate to draw the image at.
297     * @param y The y-coordinate to draw the image at.
298     * @throws SHADER_EXCEPTION .
299 */
300 exception tekGuiDrawImage(const TekGuiImage* image, const float
x, const float y) {
301     // load shader
302     tekBindShaderProgram(image_shader_program);
303
304     // set uniforms
305     tekBindTexture(image->texture_id, 0);
306     tekChainThrow(tekShaderUniformMat4(image_shader_program,
"projection", projection));
307     tekChainThrow(tekShaderUniformInt(image_shader_program,
"texture_sampler", 0));
308     tekChainThrow(tekShaderUniformVec2(image_shader_program,
"start_position", (vec2){x, y}));
309
310     // draw
311     tekDrawMesh(&image->mesh);
312     return SUCCESS;
313 }
314
315 /**

```

```

316     * Delete the memory associated with a gui image.
317     * @param image The image to delete.
318     */
319 void tekGuiDeleteImage(const TekGuiImage* image) {
320     // delete the mesh.
321     tekDeleteMesh(&image->mesh);
322
323     // delete the texture.
324     tekDeleteTexture(image->texture_id);
325 }
```

tekgui/primitives.h

```

1 #pragma once
2
3 #include "../core/exception.h"
4 #include "../tekgl/mesh.h"
5 #include "../tekgl/texture.h"
6 #include <cglm/cglm.h>
7
8 typedef struct TekGuiLine {
9     TekMesh mesh;
10    vec4 color;
11 } TekGuiLine;
12
13 typedef struct TekGuiOval {
14     TekMesh mesh;
15     float inv_width;
16     float inv_height;
17     float min_dist;
18     vec2 center;
19     vec4 color;
20 } TekGuiOval;
21
22 typedef struct TekGuiImage {
23     TekMesh mesh;
24     uint texture_id;
25 } TekGuiImage;
26
27 exception tekGuiCreateLine(vec2 point_a, vec2 point_b, float
thickness, vec4 color, TekGuiLine* line);
28 exception tekGuiDrawLine(const TekGuiLine* line);
29 void tekGuiDeleteLine(const TekGuiLine* line);
30
```

```

31 exception tekGuiCreateOval(vec2 point_a, vec2 point_b, float
thickness, flag fill, vec4 color, TekGuiOval* oval);
32 exception tekGuiDrawOval(const TekGuiOval* oval);
33 void tekGuiDeleteOval(const TekGuiOval* oval);
34
35 exception tekGuiCreateImage(float width, float height, const
char* texture_filename, TekGuiImage* image);
36 exception tekGuiDrawImage(const TekGuiImage* image, float x,
float y);
37 void tekGuiDeleteImage(const TekGuiImage* image);

```

tekgui/tekgui.c

```

1 #include "tekgui.h"
2
3 #include <stdio.h>
4 #include <string.h>
5
6 #include "../tekgl/manager.h"
7 #include "../core/yml.h"
8
9 #define NOT_INITIALISED 0
10#define INITIALISED 1
11#define DE_INITIALISED 2
12
13 static flag tek_gui_init = NOT_INITIALISED;
14 static flag tek_gui_gl_init = NOT_INITIALISED;
15 static YmlFile options_yml;
16
17 static struct TekGuiWindowDefaults window_defaults;
18 static struct TekGuiListWindowDefaults list_window_defaults;
19 static struct TekGuiTextButtonDefaults text_button_defaults;
20 static struct TekGuiTextInputDefaults text_input_defaults;
21 static TekBitmapFont default_font;
22
23 /**
24 * Get a colour from the options.yml file.
25 * @param option_name The name of the option where the colour
is stored.
26 * @param colour_name The name of the colour.
27 * @param colour The colour that is contained there.
28 * @throws YML_EXCEPTION if either of the keys does not exist
or does not point to a colour.
29 */

```

```

30 static exception tekGuiGetOptionsColour(const char*
option_name, const char* colour_name, vec4 colour) {
31     // this code demonstrates how boilerplatey my yml code is
lmao
32     // just assume that it has an r g b and a component
33     // and just jam all that into a vec4
34     YmlData* yml_data;
35     double yml_float;
36
37     tekChainThrow(ymlGet(&options_yml, &yml_data, option_name,
colour_name, "r"));
38     tekChainThrow(ymlDataToFloat(yml_data, &yml_float));
39     colour[0] = (float)yml_float;
40
41     tekChainThrow(ymlGet(&options_yml, &yml_data, option_name,
colour_name, "g"));
42     tekChainThrow(ymlDataToFloat(yml_data, &yml_float));
43     colour[1] = (float)yml_float;
44
45     tekChainThrow(ymlGet(&options_yml, &yml_data, option_name,
colour_name, "b"));
46     tekChainThrow(ymlDataToFloat(yml_data, &yml_float));
47     colour[2] = (float)yml_float;
48
49     tekChainThrow(ymlGet(&options_yml, &yml_data, option_name,
colour_name, "a"));
50     tekChainThrow(ymlDataToFloat(yml_data, &yml_float));
51     colour[3] = (float)yml_float;
52
53     return SUCCESS;
54 }
55
56 /**
57 * Load the default values for a window into a struct. Has
default values if nothing specified in options.yml
58 * @param defaults A pointer to an empty TekGuiWindowDefaults
struct.
59 * @throws YML_EXCEPTION if the yml is invalid.
60 */
61 static exception tekGuiLoadWindowDefaults(struct
TekGuiWindowDefaults* defaults) {
62     YmlData* yml_data;
63

```

```

64      // if no values provided, give benefit of doubt and provide
some defaults
65      const exception tek_exception = ymlGet(&options_yml,
&yml_data, "window_defaults");
66      if (tek_exception == YML_EXCEPTION) {
67          tekGuiLog("Missing window defaults section in
'options.yml'.");
68          defaults->x_pos = 0;
69          defaults->y_pos = 0;
70          defaults->width = 0;
71          defaults->height = 0;
72          defaults->title_width = 0;
73          defaults->border_width = 0;
74          glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>background_colour);
75          glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>border_colour);
76          glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>title_colour);
77          return SUCCESS;
78      }
79
80      // load values from options yml
81      long yml_integer;
82
83      // starting x position
84      tekChainThrow(ymlGet(&options_yml, &yml_data,
"window_defaults", "x_pos"));
85      tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
86      defaults->x_pos = (uint)yml_integer;
87
88      // starting y position
89      tekChainThrow(ymlGet(&options_yml, &yml_data,
"window_defaults", "y_pos"));
90      tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
91      defaults->y_pos = (uint)yml_integer;
92
93      // width
94      tekChainThrow(ymlGet(&options_yml, &yml_data,
"window_defaults", "width"));
95      tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
96      defaults->width = (uint)yml_integer;
97

```

```

98     // height
99     tekChainThrow(ymlGet(&options_yml, &yml_data,
100    "window_defaults", "height"));
100    tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
101    defaults->height = (uint)yml_integer;
102
103    // height of title (width?)
104    tekChainThrow(ymlGet(&options_yml, &yml_data,
105    "window_defaults", "title_width"));
105    tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
106    defaults->title_width = (uint)yml_integer;
107
108    // width of border
109    tekChainThrow(ymlGet(&options_yml, &yml_data,
110    "window_defaults", "border_width"));
110    tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
111    defaults->border_width = (uint)yml_integer;
112
113    // title text
114    tekChainThrow(ymlGet(&options_yml, &yml_data,
115    "window_defaults", "title"));
115    tekChainThrow(ymlDataToString(yml_data, &defaults->title));
116
117    // colours
118    tekChainThrow(tekGuiGetOptionsColour("window_defaults",
119    "background_colour", defaults->background_colour));
119    tekChainThrow(tekGuiGetOptionsColour("window_defaults",
120    "border_colour", defaults->border_colour));
120    tekChainThrow(tekGuiGetOptionsColour("window_defaults",
121    "title_colour", defaults->title_colour));
121
122    return SUCCESS;
123 }
124
125 /**
126  * Load the default values from options.yml into a struct. Has
127  * default values if nothing specified in options.yml
128  * @param defaults A pointer to a TekGuiListWindowDefaults
129  * struct.
130  * @throws YML_EXCEPTION if the yml file is invalid.
131  */
130 static exception tekGuiLoadListWindowDefaults(struct
TekGuiListWindowDefaults* defaults) {

```

```

131     YmlData* yml_data;
132
133     // if no defaults found, give benefit of doubt and provide
134     // some default values
135     const exception tek_exception = ymlGet(&options_yml,
136     &yml_data, "list_window_defaults");
137     if (tek_exception == YML_EXCEPTION) {
138         tekGuiLog("Missing list window defaults section in
139         'options.yml'.");
140         defaults->text_size = 0;
141         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
142         >text_colour);
143         return SUCCESS;
144     }
145
146     // otherwise load defaults
147     long yml_integer;
148
149     // text height
150     tekChainThrow(ymlGet(&options_yml, &yml_data,
151     "list_window_defaults", "text_size"));
152     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
153     defaults->text_size = (uint)yml_integer;
154
155     // maximum number of text entries to display.
156     tekChainThrow(ymlGet(&options_yml, &yml_data,
157     "list_window_defaults", "num_visible"));
158     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
159     defaults->num_visible = (uint)yml_integer;
160
161     // text colour
162
163     tekChainThrow(tekGuiGetOptionsColour("list_window_defaults",
164     "text_colour", defaults->text_colour));
165 }
166
167 /**
168  * Load the default values for a text button from the
169  * options.yml file. Has some default values if none found.
170  * @param defaults A pointer to a TekGuiTextButtonDefaults
171  * struct that is filled with the data.
172  * @throws YML_EXCEPTION if the yml file is invalid.
173 */

```

```

164 static exception tekGuiLoadTextButtonDefaults(struct
TekGuiTextButtonDefaults* defaults) {
165     YmlData* yml_data;
166
167     // attemp to get yml data, if it does not exist then give
benefit of doubt and provide some defaults
168     const exception tek_exception = ymlGet(&options_yml,
&yml_data, "text_button_defaults");
169     if (tek_exception == YML_EXCEPTION) {
170         tekGuiLog("Missing text button defaults section in
'options.yml'.");
171         defaults->x_pos = 0;
172         defaults->y_pos = 0;
173         defaults->width = 0;
174         defaults->height = 0;
175         defaults->border_width = 0;
176         defaults->text_height = 0;
177         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>background_colour);
178         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>selected_colour);
179         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>border_colour);
180         return SUCCESS;
181     }
182
183     // otherwise read defaults
184     long yml_integer;
185
186     // x position
187     tekChainThrow(ymlGet(&options_yml, &yml_data,
"text_button_defaults", "x_pos"));
188     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
189     defaults->x_pos = (uint)yml_integer;
190
191     // y position
192     tekChainThrow(ymlGet(&options_yml, &yml_data,
"text_button_defaults", "y_pos"));
193     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
194     defaults->y_pos = (uint)yml_integer;
195
196     // width

```

```

197     tekChainThrow(ymlGet(&options_yml, &yml_data,
198 "text_button_defaults", "width"));
199     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
200     defaults->width = (uint)yml_integer;
201
202     // height of button
203     tekChainThrow(ymlGet(&options_yml, &yml_data,
204 "text_button_defaults", "height"));
205     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
206     defaults->height = (uint)yml_integer;
207
208     // width of border
209     tekChainThrow(ymlGet(&options_yml, &yml_data,
210 "text_button_defaults", "border_width"));
211     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
212     defaults->border_width = (uint)yml_integer;
213
214     // height of text
215     tekChainThrow(ymlGet(&options_yml, &yml_data,
216 "text_button_defaults", "text_height"));
217     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
218     defaults->text_height = (uint)yml_integer;
219
220     // colours
221
222     tekChainThrow(tekGuiGetOptionsColour("text_button_defaults",
223 "background_colour", defaults->background_colour));
224
225     tekChainThrow(tekGuiGetOptionsColour("text_button_defaults",
226 "selected_colour", defaults->selected_colour));
227
228     tekChainThrow(tekGuiGetOptionsColour("text_button_defaults",
229 "border_colour", defaults->border_colour));
230
231     return SUCCESS;
232 }
233
234 /**
235  * Load the text input default values from the options.yml
236  * file. Has some default values
237  * @param defaults The outputted default values for text input.
238  * @throws YML_EXCEPTION if there are missing defaults.
239  */

```

```

229 static exception tekGuiLoadTextInputDefaults(struct
TekGuiTextInputDefaults* defaults) {
230     YmlData* yml_data;
231
232     // if there is no section at all, we can use some default
values, give benefit of doubt
233     const exception tek_exception = ymlGet(&options_yml,
&yml_data, "text_input_defaults");
234     if (tek_exception == YML_EXCEPTION) {
235         tekGuiLog("Missing text input defaults section in
'options.yml'.");
236         defaults->x_pos = 0;
237         defaults->y_pos = 0;
238         defaults->width = 0;
239         defaults->text_height = 0;
240         defaults->border_width = 0;
241         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>background_colour);
242         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>border_colour);
243         glm_vec4_copy((vec4){1.0f, 1.0f, 1.0f, 1.0f}, defaults-
>text_colour);
244         return SUCCESS;
245     }
246
247     // otherwise we will use the values specified by the user
248     // for each item we need to go through the arduous process
of using my yml file thing
249     // need to get the yml data, convert to the correct type
and then put into the defaults
250     long yml_integer;
251
252     // starting x pos
253     tekChainThrow(ymlGet(&options_yml, &yml_data,
"text_input_defaults", "x_pos"));
254     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
255     defaults->x_pos = (uint)yml_integer;
256
257     // starting y pos
258     tekChainThrow(ymlGet(&options_yml, &yml_data,
"text_input_defaults", "y_pos"));
259     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
260     defaults->y_pos = (uint)yml_integer;

```

```

261
262     // width
263     tekChainThrow(ymlGet(&options_yml, &yml_data,
264 "text_input_defaults", "width"));
264     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
265     defaults->width = (uint)yml_integer;
266
267     // height of text
268     tekChainThrow(ymlGet(&options_yml, &yml_data,
269 "text_input_defaults", "text_height"));
270     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
271     defaults->text_height = (uint)yml_integer;
272
273     // border width
274     tekChainThrow(ymlGet(&options_yml, &yml_data,
275 "text_input_defaults", "border_width"));
276     tekChainThrow(ymlDataToInteger(yml_data, &yml_integer));
277     defaults->border_width = (uint)yml_integer;
278
279     // background colour
280     tekChainThrow(tekGuiGetOptionsColour("text_input_defaults",
281 "background_colour", defaults->background_colour));
282     tekChainThrow(tekGuiGetOptionsColour("text_input_defaults",
283 "border_colour", defaults->border_colour));
284     tekChainThrow(tekGuiGetOptionsColour("text_input_defaults",
285 "text_colour", defaults->text_colour));
286
287     return SUCCESS;
288 }
289 /**
290 * Get the default values for a window gui element.
291 * @param defaults The outputted default values.
292 * @throws FAILURE if TekGUI is not initialised.
293 */
294 exception tekGuiGetWindowDefaults(struct TekGuiWindowDefaults*
295 defaults) {
296     if (!tek_gui_init)
297         tekThrow(FAILURE, "TekGUI is not initialised.")
298
299     // copy over the values.
300     memcpy(defaults, &window_defaults, sizeof(struct
301 TekGuiWindowDefaults));

```

```

296     return SUCCESS;
297 }
298 /**
299  * Get the default values for a list window.
300  * @param defaults The outputted default values.
301  * @throws FAILURE if tek gui not initialised.
302  */
303 exception tekGuiGetListWindowDefaults(struct
TekGuiListWindowDefaults* defaults) {
304     if (!tek_gui_init)
305         tekThrow(FAILURE, "TekGUI is not initialised.")
306
307     // copy over values
308     memcpy(defaults, &list_window_defaults, sizeof(struct
TekGuiListWindowDefaults));
309     return SUCCESS;
310 }
311
312 /**
313  * Get default values for tek gui button element.
314  * @param defaults The outputted default values.
315  * @throws FAILURE if tek gui is not initialised.
316  */
317 exception tekGuiGetTextButtonDefaults(struct
TekGuiTextButtonDefaults* defaults) {
318     if (!tek_gui_init)
319         tekThrow(FAILURE, "TekGUI is not initialised.")
320
321     // copy over the values.
322     memcpy(defaults, &text_button_defaults, sizeof(struct
TekGuiTextButtonDefaults));
323     return SUCCESS;
324 }
325
326 /**
327  * Get default values for text input gui elements.
328  * @param defaults The outputted default values.
329  * @throws FAILURE if tek gui is not initialised.
330  */
331 exception tekGuiGetTextInputDefaults(struct
TekGuiTextInputDefaults* defaults) {
332     if (!tek_gui_init)

```

```

334         tekThrow(FAILURE, "TekGUI is not initialised.")
335
336     // copy over values that have been loaded already.
337     memcpy(defaults, &text_input_defaults, sizeof(struct
TekGuiTextInputDefaults));
338     return SUCCESS;
339 }
340
341 /**
342 * Delete callback for tek gui code.
343 */
344 void tekGuiDelete() {
345     // delete options yml
346     ymlDelete(&options_yml);
347     tek_gui_init = DE_INITIALISED;
348     tek_gui_gl_init = DE_INITIALISED;
349 }
350
351 /**
352 * Callback for when opengl has loaded for tek gui code.
353 * @throws FREETYPE_EXCEPTION if could not load default font.
354 */
355 exception tekGuiGLLoad() {
356     // default font is hard coded... lol
357     tekChainThrow(tekCreateBitmapFont("../res/urwgothic.ttf",
358     0, 64, &default_font));
359     tek_gui_gl_init = INITIALISED;
360     return SUCCESS;
361 }
362 /**
363 * Initialisation function for tek gui base / loader code.
364 */
365 tek_init tekGuiInit() {
366     // reading options yml
367     exception tek_exception =
ymlReadFile("../tekgui/options.yml", &options_yml);
368     if (tek_exception) return;
369
370     // setup callbacks
371     tek_exception = tekAddDeleteFunc(tekGuiDelete);
372     if (tek_exception) return;
373 }
```

```

374     tek_exception = tekAddGLLoadFunc(tekGuiGLLoad);
375     if (tek_exception) return;
376
377     // load defaults for different areas
378     // also display if init failed, likely that people could
mess with these config files.
379     tek_exception = tekGuiLoadWindowDefaults(&window_defaults);
380     if (tek_exception) {
381         tekGuiLog("Error during TekGui Init: %d\n",
tek_exception);
382         tekPrintException();
383         return;
384     }
385
386     tek_exception =
tekGuiLoadListWindowDefaults(&list_window_defaults);
387     if (tek_exception) {
388         tekGuiLog("Error during TekGui Init: %d\n",
tek_exception);
389         tekPrintException();
390         return;
391     }
392
393     tek_exception =
tekGuiLoadTextButtonDefaults(&text_button_defaults);
394     if (tek_exception) {
395         tekGuiLog("Error during TekGui Init: %d\n",
tek_exception);
396         tekPrintException();
397         return;
398     }
399
400     tek_exception =
tekGuiLoadTextInputDefaults(&text_input_defaults);
401     if (tek_exception) {
402         tekGuiLog("Error during TekGui Init: %d\n",
tek_exception);
403         tekPrintException();
404         return;
405     }
406
407     // if worked without errors, then we are initialised
408     tek_gui_init = INITIALISED;

```

```

409 }
410 /**
411 * Get the default font for the program.
412 * @param font The outputted default font.
413 * @throws OPENGL_EXCEPTION if opengl not initialised.
414 */
415 exception tekGuiGetDefaultFont(TekBitmapFont** font) {
416     if (tek_gui_gl_init != INITIALISED)
417         tekThrow(OPENGL_EXCEPTION, "Attempted to run function
before OpenGL initialised.");
418
419     // literally what it says on the tin
420     *font = &default_font;
421     return SUCCESS;
422 }

```

tekgui/tekgui.h

```

1 #pragma once
2
3 #include <cglm/vec4.h>
4 #include "../core/exception.h"
5 #include "../tekgl/font.h"
6
7 struct TekGuiWindowDefaults {
8     uint x_pos;
9     uint y_pos;
10    uint width;
11    uint height;
12    uint title_width;
13    uint border_width;
14    vec4 background_colour;
15    vec4 border_colour;
16    vec4 title_colour;
17    char* title;
18 };
19
20 struct TekGuilistWindowDefaults {
21     uint text_size;
22     vec4 text_colour;
23     uint num_visible;
24 };
25

```

```

26 struct TekGuiTextButtonDefaults {
27     uint x_pos;
28     uint y_pos;
29     uint width;
30     uint height;
31     uint border_width;
32     uint text_height;
33     vec4 background_colour;
34     vec4 selected_colour;
35     vec4 border_colour;
36 };
37
38 struct TekGuiTextInputDefaults {
39     uint x_pos;
40     uint y_pos;
41     uint width;
42     uint text_height;
43     uint border_width;
44     vec4 background_colour;
45     vec4 border_colour;
46     vec4 text_colour;
47 };
48
49 /**
50 * Mostly unused logging function. Just prints [INFO] TekGui:
before the message
51 */
52 #define tekGuiLog(...) printf("[INFO] TekGui: ");
printf(__VA_ARGS__)
53
54 exception tekGuiGetWindowDefaults(struct TekGuiWindowDefaults*
defaults);
55 exception tekGuiGetListWindowDefaults(struct
TekGuiListWindowDefaults* defaults);
56 exception tekGuiGetTextButtonDefaults(struct
TekGuiTextButtonDefaults* defaults);
57 exception tekGuiGetTextInputDefaults(struct
TekGuiTextInputDefaults* defaults);
58 exception tekGuiGetDefaultFont(TekBitmapFont** font);

```

tekgui/text_button.c

```

1 #include "text_button.h"
2

```

```

3 #include <cglm/vec2.h>
4
5 #include "tekgui.h"
6 #include "../core/vector.h"
7
8 #include "../tekgl/manager.h"
9 #include "../tekgl/shader.h"
10 #include "glad/glad.h"
11
12 /**
13  * Get the dat needed to draw a box for the button.
14  * @param button The text button to get the box data for.
15  * @param box_data The outputted box data
16  */
17 static void tekGuiGetTextButtonData(const TekGuiTextButton*
button, TekGuiBoxData* box_data) {
18     // minmax = minimum and maximum extents in an axis
19     // minmax_i = internal box, excluding border
20     glm_vec2_copy((vec2){(float)button->button.hitbox_x,
21 (float)(button->button.hitbox_x + button->button.hitbox_width)}, 
box_data->minmax_x);
22     glm_vec2_copy((vec2){(float)button->button.hitbox_y,
23 (float)(button->button.hitbox_y + button->button.hitbox_height)}, 
box_data->minmax_y);
24     glm_vec2_copy((vec2){(float)(button->button.hitbox_x +
button->border_width), (float)(button->button.hitbox_x + button-
>button.hitbox_width - button->border_width)}, box_data->minmax_ix);
25     glm_vec2_copy((vec2){(float)(button->button.hitbox_y +
button->border_width), (float)(button->button.hitbox_y + button-
>button.hitbox_height - button->border_width)}, box_data-
>minmax_iy);
26 }
27 /**
28  * Add box mesh to the list of boxes to be drawn. All box
backgrounds are stored in the same vertex buffer
29  * So each box usage needs to be registered to find the index
of the box in this buffer.
30  * @param text_button The text button to add a mesh for.
31  * @param index The outputted index of the box.
32  * @throws OPENGL_EXCEPTION .
33 */

```

```

33 static exception tekGuiTextButtonAddGLMesh(const
TekGuiTextButton* text_button, uint* index) {
34     // get box data and create new box and add to list of
boxes.
35     TekGuiBoxData box_data = {};
36     tekGuiGetTextButtonData(text_button, &box_data);
37     tekChainThrow(tekGuiCreateBox(&box_data, index));
38     return SUCCESS;
39 }
40
41 /**
42 * Update the background box mesh of the text button.
43 * @param text_button The text button to update the mesh for.
44 * @throws OPENGL_EXCEPTION .
45 */
46 static exception tekGuiTextButtonUpdateGLMesh(const
TekGuiTextButton* text_button) {
47     // get box data and update
48     TekGuiBoxData box_data = {};
49     tekGuiGetTextButtonData(text_button, &box_data);
50     tekChainThrow(tekGuiUpdateBox(&box_data, text_button-
>mesh_index));
51     return SUCCESS;
52 }
53
54 /**
55 * Create some text to be displayed on the text button.
56 * @param button The button to create the text for.
57 * @throws OPENGL_EXCEPTION .
58 */
59 static exception tekGuiTextButtonCreateText(TekGuiTextButton*
button) {
60     // get the default font.
61     TekBitmapFont* font;
62     tekChainThrow(tekGuiGetDefaultFont(&font));
63
64     // create text with button data
65     tekChainThrow(tekCreateText(button->text, button-
>text_height, font, &button->tek_text));
66     return SUCCESS;
67 }
68
69 /**

```

```

70  * Recreate the displayed text on a text button.
71  * @param button The button to recreate the text for.
72  * @throws OPENGL_EXCEPTION .
73  */
74 static exception tekGuiTextButtonRecreateText(TekGuiTextButton*
button) {
75     // delete old text and make a new one
76     tekDeleteText(&button->tek_text);
77     tekChainThrow(tekGuiTextButtonCreateText(button));
78     return SUCCESS;
79 }
80
81 /**
82 * Callback for text buttons. Passes on the callback to the
text button and update hovered status.
83 * @param button The base button that received the callback.
84 * @param callback_data The callback data associated with the
button press.
85 */
86 static void tekGuiTextButtonCallback(TekGuiButton* button,
TekGuiButtonCallbackData callback_data) {
87     // get text button from button data.
88     TekGuiTextButton* text_button = (TekGuiTextButton*)button-
>data;
89
90     // update hovered status
91     switch (callback_data.type) {
92         case TEK_GUI_BUTTON_MOUSE_ENTER_CALLBACK:
93             text_button->hovered = 1;
94             break;
95         case TEK_GUI_BUTTON_MOUSE_LEAVE_CALLBACK:
96             text_button->hovered = 0;
97             break;
98         default:
99             break;
100    }
101
102    // pass callback onto the text button callback
103    if (text_button->callback) text_button-
>callback(text_button, callback_data);
104 }
105
106 /**

```

```

107  * Create a new text button which is a button that has text.
108  * @param text The text on the button.
109  * @param button The outputted button.
110  * @throws MEMORY_EXCEPTION if malloc() fails.
111  */
112 exception tekGuiCreateTextButton(const char* text,
TekGuiTextButton* button) {
113     // collect default values for a text button e.g. size and
colours.
114     struct TekGuiTextButtonDefaults defaults = {};
115     tekChainThrow(tekGuiGetTextButtonDefaults(&defaults));
116
117     // create the internal button that will receive events etc.
118     tekChainThrow(tekGuiCreateButton(&button->button));
119     tekGuiSetButtonPosition(&button->button, defaults.x_pos,
defaults.y_pos);
120     tekGuiSetButtonSize(&button->button, defaults.width,
defaults.height);
121     button->button.callback = tekGuiTextButtonCallback;
122     button->border_width = defaults.border_width;
123     button->text_height = defaults.text_height;
124     button->hovered = 0;
125     button->button.data = button;
126     button->callback = NULL;
127     glm_vec4_copy(defaults.background_colour, button-
>background_colour);
128     glm_vec4_copy(defaults.selected_colour, button-
>selected_colour);
129     glm_vec4_copy(defaults.border_colour, button-
>border_colour);
130
131     // fill up text buffer with the text to be displayed.
132     const uint len_text = strlen(text) + 1;
133     button->text = (char*)malloc(len_text * sizeof(char));
134     if (!button->text)
135         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for text buffer.");
136     memcpy(button->text, text, len_text);
137
138     // generate the tek_text struct.
139     tekChainThrow(tekGuiTextButtonCreateText(button));
140
141     // generate the mesh to display the button.

```

```

142     tekChainThrow(tekGuiTextButtonAddGLMesh(button, &button-
>mesh_index));
143
144     return SUCCESS;
145 }
146
147 /**
148 * Set the pixel position of a text button on the window.
149 * @param button The button to set the position of.
150 * @param x_pos The x position to set the button to.
151 * @param y_pos The y position to set the button to.
152 * @throws OPENGL_EXCEPTION .
153 */
154 exception tekGuiSetTextButtonPosition(TekGuiTextButton* button,
const uint x_pos, const uint y_pos) {
155     // set position and recreate mesh
156     tekGuiSetButtonPosition(&button->button, x_pos, y_pos);
157     tekChainThrow(tekGuiTextButtonUpdateGLMesh(button));
158     return SUCCESS;
159 }
160
161 /**
162 * Set the size of a text button in pixels.
163 * @param button The button to update the size of.
164 * @param width The new width of the button in pixels.
165 * @param height The new height of the button in pixels.
166 * @throws OPENGL_EXCEPTION .
167 */
168 exception tekGuiSetTextButtonSize(TekGuiTextButton* button,
const uint width, const uint height) {
169     // update button size and recreate mesh
170     tekGuiSetButtonSize(&button->button, width, height);
171     tekChainThrow(tekGuiTextButtonUpdateGLMesh(button));
172     tekChainThrow(tekGuiTextButtonRecreateText(button));
173     return SUCCESS;
174 }
175
176 /**
177 * Update the text that is displayed on a text button.
178 * @param button The button to update the text for.
179 * @param text The new text to display.
180 * @throws MEMORY_EXCEPTION if malloc() fails.
181 */

```

```

182 exception tekGuiSetTextButtonText(TekGuiTextButton* button,
183 const char* text) {
184     // free the pointer to the text buffer if it exists.
185     if (button->text)
186         free(button->text);
187
188     // create new buffer for the new text to be stored in.
189     const uint len_text = strlen(text) + 1;
190     button->text = (char*)malloc(len_text * sizeof(char));
191     if (!button->text)
192         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for text buffer.");
193
194     // copy over the new text.
195     memcpy(button->text, text, len_text);
196
197     // rebuild tek_text struct.
198     tekGuiTextButtonRecreateText(button);
199
200     return SUCCESS;
201 }
202 /**
203 * Draw a text button to the screen, drawing the background and
text. Also shade the button if hovered.
204 * @param button The button to draw to the screen.
205 * @throws OPENGL_EXCEPTION .
206 */
207 exception tekGuiDrawTextButton(const TekGuiTextButton* button)
{
208     // get colour based on if hovered or not
209     vec4 background_colour;
210     if (button->hovered) {
211         memcpy(background_colour, button->selected_colour,
212             sizeof(vec4));
213     } else {
214         memcpy(background_colour, button->background_colour,
215             sizeof(vec4));
216     }
217
218     // draw the background of the button
219     tekChainThrow(tekGuiDrawBox(button->mesh_index,
background_colour, button->border_colour));

```

```

218
219     // draw the text + slight offset to center the text.
220     const float x = (float)button->button.hitbox_x +
((float)button->button.hitbox_width - button->tek_text.width) *
0.5f;
221     const float y = (float)button->button.hitbox_y +
((float)button->button.hitbox_height - button->tek_text.height) *
0.5f;
222     tekChainThrow(tekDrawText(&button->tek_text, x, y));
223
224     return SUCCESS;
225 }
226
227 /**
228 * Delete a text button, freeing any memory allocated.
229 * @param button
230 */
231 void tekGuiDeleteTextButton(const TekGuiTextButton* button) {
232     // free displayed text
233     if (button->text)
        free(button->text);
235
236     // delete button and text gui elements
237     tekGuiDeleteButton(&button->button);
238     tekDeleteText(&button->tek_text);
239 }
```

tekgui/text_button.h

```

1 #pragma once
2
3 #include "button.h"
4 #include <cglm/vec4.h>
5 #include "../tekgl/text.h"
6 #include "box_manager.h"
7
8 struct TekGuiTextButton;
9
10 typedef void(*TekGuiTextButtonCallback)(struct TekGuiTextButton*
button, TekGuiButtonCallbackData callback_data);
11
12 typedef struct TekGuiTextButton {
13     TekGuiButton button;
14     char* text;
```

```

15     TekText tek_text;
16     uint border_width;
17     uint text_height;
18     vec4 background_colour;
19     vec4 selected_colour;
20     vec4 border_colour;
21     uint mesh_index;
22     flag hovered;
23     TekGuiTextButtonCallback callback;
24     void* data;
25 } TekGuiTextButton;
26
27 exception tekGuiCreateTextButton(const char* text,
TekGuiTextButton* button);
28 exception tekGuiSetTextButtonPosition(TekGuiTextButton* button,
uint x_pos, uint y_pos);
29 exception tekGuiSetTextButtonSize(TekGuiTextButton* button, uint
width, uint height);
30 exception tekGuiSetTextButtonText(TekGuiTextButton* button,
const char* text);
31 exception tekGuiDrawTextButton(const TekGuiTextButton* button);
32 void tekGuiDeleteTextButton(const TekGuiTextButton* button);

```

tekgui/text_input.c

```

1 #include "text_input.h"
2
3 #include "../tekgl/manager.h"
4 #include "box_manager.h"
5 #include "tekgui.h"
6 #include "glad/glad.h"
7 #include <GLFW/glfw3.h>
8 #include <time.h>
9
10 #define NOT_INITIALISED 0
11 #define INITIALISED 1
12 #define DE_INITIALISED 2
13
14 #define CURSOR '|'
15
16 static TekGuiTextInput* selected_input = 0;
17 static TekBitmapFont monospace_font = {};
18
19 /**

```

```

20  * Get the data for drawing a background box from a text input.
21  * @param text_input The text input to get the data from.
22  * @param box_data An empty box data struct to be filled with
the data.
23  */
24 static void tekGuiGetTextInputData(const TekGuiTextInput*
text_input, TekGuiBoxData* box_data) {
25     // minmax = the minimum and maximum extent of that
direction
26     // i = internal, this is the actual drawn area without
borders
27     glm_vec2_copy((vec2){(float)text_input->button.hitbox_x,
(float)text_input->button.hitbox_x + (float)text_input-
>button.hitbox_width}, box_data->minmax_x);
28     glm_vec2_copy((vec2){(float)text_input->button.hitbox_y,
(float)text_input->button.hitbox_y + (float)text_input-
>button.hitbox_height}, box_data->minmax_y);
29     glm_vec2_copy((vec2){(float)text_input->button.hitbox_x +
(float)text_input->border_width, (float)text_input->button.hitbox_x
+ (float)(text_input->button.hitbox_width - text_input-
>border_width)}, box_data->minmax_ix);
30     glm_vec2_copy((vec2){(float)text_input->button.hitbox_y +
(float)text_input->border_width, (float)text_input->button.hitbox_y
+ (float)(text_input->button.hitbox_height - text_input-
>border_width)}, box_data->minmax_iy);
31 }
32
33 /**
34  * Get the number of characters currently being displayed to
the screen by a text input.
35  * @param text_input The text input to get the length of.
36  * @return The number of characters being displayed to the
screen.
37 */
38 static uint tekGuiGetTextInputTextLength(const TekGuiTextInput*
text_input) {
39     // if a max length has been found, return it
40     if (text_input->text_max_length != -1) {
41         return (uint)text_input->text_max_length;
42     }
43
44     // otherwise, not yet reached max, return current length.
45     return text_input->text.length;

```

```

46 }
47
48 /**
49 * Move the cursor backwards, and scroll the text if needed.
50 * @param text_input The text input to decrement the cursor of.
51 */
52 static void tekGuiTextInputDecrementCursor(TekGuiTextInput*
text_input) {
53     // if not already reached the start, decrement position
54     if (text_input->cursor_index > 0)
55         text_input->cursor_index--;
56     else
57         return;
58
59     // move the text forwards if scrolling would move the
cursor off the end
60     if (text_input->text_start_index > 0 && text_input-
>cursor_index < text_input->text_start_index)
61         text_input->text_start_index--;
62 }
63
64 /**
65 * Move the cursor forwards by a single position, scrolling the
text if needed.
66 * @param text_input The text input to increment the cursor of.
67 */
68 static void tekGuiTextInputIncrementCursor(TekGuiTextInput*
text_input) {
69     // only increment if there is space for the cursor to move
into.
70     if (text_input->cursor_index + 2 < text_input->text.length)
71         text_input->cursor_index++;
72     else
73         return;
74
75     // if max length not achieved, then last step can be
skipped
76     if (text_input->text_max_length <= -1)
77         return;
78
79     // if cursor would be going off the end, scroll all the
text back to fit the next char on the end

```

```

80      if ((int)text_input->cursor_index + 1 > text_input-
>text_start_index + text_input->text_max_length) {
81          text_input->text_start_index++;
82      }
83  }
84
85 /**
86  * Create a new text input box background shape for a new text
87  * input.
88  * @param text_input The text input to create the background
89  * for.
90  * @param index The outputted index of the box. All the box
91  * meshes are stored in a single vertex buffer.
92  * @throws OPENGL_EXCEPTION .
93  */
94 static exception tekGuiTextInputAddGLMesh(const
TekGuiTextInput* text_input, uint* index) {
95     // basic box background creation
96     TekGuiBoxData box_data = {};
97     tekGuiGetTextInputData(text_input, &box_data);
98     tekChainThrow(tekGuiCreateBox(&box_data, index));
99     return SUCCESS;
100 }
101 /**
102  * Visually update the text input background area.
103  * @param text_input The text input to update.
104  * @throws OPENGL_EXCEPTION .
105  */
106 static exception tekGuiTextInputUpdateGLMesh(const
TekGuiTextInput* text_input) {
107     // basic box background
108     TekGuiBoxData box_data = {};
109     tekGuiGetTextInputData(text_input, &box_data);
110     tekChainThrow(tekGuiUpdateBox(&box_data, text_input-
>mesh_index));
111     return SUCCESS;
112 }
113 /**
114  * Allocate a buffer of the visible text being displayed.
115  * @param text_input The text to get a pointer of.

```

```

115  * @param buffer A pointer to where the buffer should be
allocated.
116  * @throws MEMORY_EXCEPTION if malloc() fails.
117  */
118 static exception tekGuiTextInputGetTextPtr(const
TekGuiTextInput* text_input, char** buffer) {
119      // allocate sum memory
120      const uint length =
tekGuiGetTextInputTextLength(text_input);
121      *buffer = (char*)malloc(length * sizeof(char));
122      if (!*buffer)
123          tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for text buffer.");
124
125      // if its possible that some text is cut off, then dont
copy it all
126      if (text_input->text_max_length != -1) {
127          memcpy(*buffer, (char*)text_input->text.internal +
text_input->text_start_index, length);
128          (*buffer)[length - 1] = 0;
129          return SUCCESS;
130      }
131
132      // if text not that long yet, then copy it straight over
133      memcpy(*buffer, text_input->text.internal, length);
134      return SUCCESS;
135 }
136
137 /**
138  * Create the text mesh to be displayed to the user.
139  * @param text_input The text input to create the text for.
140  * @throws OPENGL_EXCEPTION .
141  */
142 static exception tekGuiTextInputCreateText(TekGuiTextInput*
text_input) {
143     char* text = 0;
144
145     // create the text
146     tekChainThrow(tekGuiTextInputGetTextPtr(text_input,
&text));
147     tekChainThrow(tekCreateText(text, text_input->text_height,
&monospace_font, &text_input->tek_text));
148

```

```

149     // if too long, recreate with less characters
150     const float max_width = (float)text_input-
>button.hitbox_width - 2 * text_input->tek_text.height;
151     if (text_input->text_max_length == -1 && text_input-
>tek_text.width >= max_width) {
152         text_input->text_max_length = (int)text_input-
>text.length - 1;
153         tekChainThrow(tekCreateText(text, text_input-
>text_height, &monospace_font, &text_input->tek_text));
154     }
155
156     // create version that includes the cursor
157     const uint cursor_index = text_input->cursor_index -
text_input->text_start_index;
158     text[cursor_index] = CURSOR;
159     tekChainThrow(tekCreateText(text, text_input->text_height,
&monospace_font, &text_input->tek_text_cursor));
160
161     free(text);
162     return SUCCESS;
163 }
164
165 /**
166 * Recreate the meshes needed to draw the text so that changes
to the contents of the text input are reflected visually.
167 * @param text_input The text input to recreate.
168 * @throws OPENGL_EXCEPTION .
169 */
170 static exception tekGuiTextInputRecreateText(TekGuiTextInput*
text_input) {
171     char* text = 0;
172
173     // get text pointer.
174     // points to where the text is started to be displayed.
175     tekChainThrow(tekGuiTextInputGetTextPtr(text_input,
&text));
176     tekChainThrow(tekUpdateText(&text_input->tek_text, text,
text_input->text_height));
177
178     // check if the entire text can fit on or nah
179     const float max_width = (float)text_input-
>button.hitbox_width - 2 * text_input->tek_text.height;

```

```

180     if (text_input->text_max_length == -1 && text_input-
181         >tek_text.width >= max_width) {
182         text_input->text_max_length = (int)text_input-
183         >text.length - 1;
184         tekChainThrow(tekUpdateText(&text_input->tek_text,
185             text, text_input->text_height));
186     }
187
188     // create another version of the text that includes the
189     // cursor icon.
190     const uint cursor_index = text_input->cursor_index -
191         text_input->text_start_index;
192     text[cursor_index] = CURSOR;
193     tekChainThrow(tekUpdateText(&text_input->tek_text_cursor,
194         text, text_input->text_height));
195
196     free(text);
197     return SUCCESS;
198 }
199
200 /**
201 * Add a character to the text input at the cursor position.
202 * @param text_input The text input to add the character to.
203 * @param codepoint The character to add to the text input.
204 * @throws VECTOR_EXCEPTION .
205 */
206 static exception tekGuiTextInputAdd(TekGuiTextInput*
207 text_input, const char codepoint) {
208     // add the character
209     tekChainThrow(vectorInsertItem(&text_input->text,
210         text_input->cursor_index, &codepoint));
211
212     // move the cursor forwards
213     tekGuiTextInputIncrementCursor(text_input);
214
215     return SUCCESS;
216 }
217
218 /**
219 * Remove a character from a text input gui.
220 * @param text_input The text input to remove a character from.
221 * @throws VECTOR_EXCEPTION .
222 */

```

```

214 static exception tekGuiTextInputRemove(TekGuiTextInput*
text_input) {
215     if (text_input->text.length <= 1) return SUCCESS;
216     if (text_input->cursor_index <= 0) return SUCCESS; // if
cursor reached the start, cannot remove now
217
218     // remove character before cursor index
219     tekChainThrow(vectorRemoveItem(&text_input->text,
text_input->cursor_index - 1, NULL));
220
221     // shift cursor back
222     if (text_input->cursor_index > 0) {
223         text_input->cursor_index--;
224         // also potentially shift the entire text back
225         if (text_input->text_start_index > 0)
226             text_input->text_start_index--;
227     }
228     return SUCCESS;
229 }
230
231 /**
232 * Callback for listening for actual typed text, so caps lock
gives capital letters and such.
233 * @param codepoint The codepoint of the last typed character.
234 */
235 static void tekGuiTextInputCharCallback(const uint codepoint) {
236     if (!selected_input) return; // if no input selected, then
ignore
237     tekGuiTextInputAdd(selected_input, (char)codepoint); // otherwise write character to the selected input.
238     tekGuiTextInputRecreateText(selected_input);
239 }
240
241 /**
242 * Finish inputting on a text input, reset the text to initial
position.
243 * @param text_input The text input to finish inputting to.
244 * @throws EXCEPTION that the further callbacks could call.
245 */
246 static exception tekGuiFinishTextInput(TekGuiTextInput*
text_input) {
247     // reset cursor position
248     selected_input = 0;

```

```

249     text_input->cursor_index = 0;
250     text_input->text_start_index = 0;
251
252     if (text_input->callback) // if there is a callback, call
it
253         tekChainThrow(text_input->callback(text_input,
text_input->text.internal, text_input->text.length));
254
255     // redraw text from new starting position
256     tekChainThrow(tekGuiTextInputRecreateText(text_input));
257     return SUCCESS;
258 }
259
260 /**
261 * Callback for when a key is pressed to affect text inputs.
Listens for enter key, arrow keys etc.
262 * @param key The key code of the key being pressed.
263 * @param scancode The scancode of the key.
264 * @param action The action of the key e.g. press, release
265 * @param mods The modifiers acting on the key.
266 */
267 static void tekGuiTextInputKeyCallback(const int key, const int
scancode, const int action, const int mods) {
268     if (!selected_input) return;
269     if (action != GLFW_RELEASE && action != GLFW_REPEAT)
return;
270
271     TekGuiTextInput* text_input = selected_input;
272
273     switch (key) {
274     case GLFW_KEY_ENTER:
275         // on enter, finish writing
276         tekGuiFinishTextInput(selected_input);
277         break;
278     case GLFW_KEY_BACKSPACE:
279         // on backspace, delete a character
280         tekGuiTextInputRemove(selected_input);
281         break;
282     case GLFW_KEY_LEFT:
283         // on key left, move the cursor backwards
284         tekGuiTextInputDecrementCursor(selected_input);
285         break;
286     case GLFW_KEY_RIGHT:

```

```

287         // on key right, move the cursor forwards
288         tekGuiTextInputIncrementCursor(selected_input);
289         break;
290     default:
291         return;
292     }
293
294     tekGuiTextInputRecreateText(text_input);
295 }
296
297 /**
298 * Callback for when the text input is clicked.
299 * @param button The internal button of the text input.
300 * @param callback_data The callback data e.g. mouse position.
301 */
302 static void tekGuiTextInputButtonCallback(TekGuiButton* button,
TekGuiButtonCallbackData callback_data) {
303     TekGuiTextInput* text_input = (TekGuiTextInput*)button-
>data; // get the text input stored in button data
304
305     switch (callback_data.type) {
306     case TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK:
307         // if not clicking then ignore
308         if (callback_data.data.mouse_button.button != GLFW_MOUSE_BUTTON_LEFT || callback_data.data.mouse_button.action != GLFW_RELEASE)
309             break;
310         if (text_input == selected_input) // if this input is
already selected, then dont re select it
311             break;
312         if (selected_input)
313             tekGuiFinishTextInput(selected_input);
314         selected_input = text_input;
315         break;
316     default:
317         break;
318     }
319 }
320
321 /**
322 * Load text input related things that require opengl. Just
loads fonts presently.
323 * @throws FREETYPE_EXCEPTION if couldn't load the font.

```

```

324     */
325 static exception tekGuiTextInputGLLoad() {
326     // commenting every function !!
327     tekChainThrow(tekCreateBitmapFont("../res/inconsolata.ttf",
328     0, 64, &monospace_font));
328     return SUCCESS;
329 }
330
331 /**
332 * Initialise some stuff for text inputs, just callbacks for
333 keys and typing and loading.
334 */
335 tek_init tekTextInputInit() {
336     // user interactivity
337     tekAddCharCallback(tekTextInputCharCallback);
338     tekAddKeyCallback(tekTextInputKeyCallback);
339
340     // graphical stuff
341     tekAddGLLoadFunc(tekTextInputGLLoad);
342 }
343 /**
344 * Create a new text input gui element.
345 * @param text_input A pointer to an existing but empty text
346 input to be filled with data.
347 * @throws OPENGL_EXCEPTION .
348 */
349 exception tekGuiCreateTextInput(TekGuiTextInput* text_input) {
350     // get defaults from the defaults file.
351     struct TekGuiTextInputDefaults defaults = {};
352     tekChainThrow(tekGuiGetTextInputDefaults(&defaults));
353
354     // create the internal button for checking when selected by
355     // mouse
356     tekChainThrow(tekGuiCreateButton(&text_input->button));
357     tekGuiSetButtonPosition(&text_input->button,
358     (int)defaults.x_pos, (int)defaults.y_pos);
359     tekGuiSetButtonSize(&text_input->button, defaults.width,
360     defaults.text_height * 5 / 4);
361     text_input->button.callback =
362     tekGuiTextInputButtonCallback;
363     text_input->button.data = (void*)text_input;

```

```

359     tekChainThrow(tekGuiTextInputAddGLMesh(text_input,
&text_input->mesh_index));
360
361     // create buffer for storing text
362     tekChainThrow(vectorCreate(16, sizeof(char), &text_input-
>text));
363     const char zero = 0;
364     tekChainThrow(vectorAddItem(&text_input->text, &zero));
365     tekChainThrow(vectorAddItem(&text_input->text, &zero));
366     text_input->text_height = defaults.text_height;
367     text_input->cursor_index = 0;
368     text_input->text_max_length = -1;
369     text_input->text_start_index = 0;
370     tekChainThrow(tekGuiTextInputCreateText(text_input));
371
372     // copy in defaults
373     text_input->border_width = defaults.border_width;
374     glm_vec4_copy(defaults.background_colour, text_input-
>background_colour);
375     glm_vec4_copy(defaults.border_colour, text_input-
>border_colour);
376     glm_vec4_copy(defaults.text_colour, text_input-
>text_colour);
377
378     return SUCCESS;
379 }
380
381 /**
382 * Draw a text input gui element to the screen.
383 * @param text_input The text input to draw
384 * @throws OPENGL_EXCEPTION .
385 */
386 exception tekGuiDrawTextInput(const TekGuiTextInput*
text_input) {
387     // draw the outer box
388     tekChainThrow(tekGuiDrawBox(text_input->mesh_index,
text_input->background_colour, text_input->border_colour));
389     // text positionm, slight offset from left edge
390     const float x = (float)(text_input->button.hitbox_x + (int)
(text_input->text_height / 2));
391     const float y = (float)text_input->button.hitbox_y;
392
393     // flickering cursor

```

```

394     // basically get the current time
395     // if in the first half of a second, show the cursor,
otherwise dont show it
396     flag display_cursor = 0;
397     struct timespec time;
398     clock_gettime(CLOCK_MONOTONIC, &time);
399     if (text_input == selected_input && time.tv_nsec % BILLION
> BILLION / 2) {
400         display_cursor = 1;
401     }
402
403     if (display_cursor) {
404         tekChainThrow(tekDrawColouredText(&text_input-
>tek_text_cursor, x, y, text_input->text_colour));
405     } else {
406         tekChainThrow(tekDrawColouredText(&text_input-
>tek_text, x, y, text_input->text_colour));
407     }
408
409     return SUCCESS;
410 }
411
412 /**
413 * Update the position of a text input gui element.
414 * @param text_input The text input to update the position of.
415 * @param x_pos The new x position of the text input.
416 * @param y_pos The new y position of the text input.
417 * @throws MEMORY_EXCEPTION .
418 */
419 exception tekGuiSetTextInputPosition(TekGuiTextInput*
text_input, int x_pos, int y_pos) {
420     tekGuiSetButtonPosition(&text_input->button, x_pos, y_pos);
// update internal button
421     tekChainThrow(tekGuiTextInputUpdateGLMesh(text_input)); // update visually.
422
423     return SUCCESS;
424 }
425
426 /**
427 * Set the width and height of a text input gui element in pixels.
428 * @param text_input The text input to change the size of.

```

```

429     * @param width The new width of the text input.
430     * @param height The new height of the text input.
431     * @throws MEMORY_EXCEPTION .
432 */
433 exception tekGuiSetTextInputSize(TekGuiTextInput* text_input,
434     uint width, uint height) {
434     tekGuiSetButtonSize(&text_input->button, width, height); // update internal button
435     tekChainThrow(tekGuiTextInputUpdateGLMesh(text_input)); // update visuals
436
437     return SUCCESS;
438 }
439
440 /**
441     * Set the text being displayed by a text input.
442     * @param text_input The text input to update with new text.
443     * @param text The new text to be stored in the input.
444     * @throws VECTOR_EXCEPTION .
445     * @throws MEMORY_EXCEPTION if malloc() fails.
446 */
447 exception tekGuiSetTextInputText(TekGuiTextInput* text_input,
448     const char* text) {
448     // clear out the old input and add two null characters at the end.
449     // first null character is where the '|' will flicker
450     // probably a better way to do it but idc
451     text_input->cursor_index = 0;
452     text_input->text_start_index = 0;
453     vectorClear(&text_input->text);
454
455     const char zero = 0;
456     for (uint i = 0; i < 2; i++)
457         tekChainThrow(vectorAddItem(&text_input->text, &zero));
458
459     // if text is null, assume an empty string is wanted.
460     if (text) {
461         const uint len_text = strlen(text);
462         for (uint i = 0; i < len_text; i++) {
463             tekChainThrow(tekGuiTextInputAdd(text_input,
463             text[i]));
464             if (text_input->text_max_length == -1)

```

```

465
tekChainThrow(tekGuiTextInputRecreateText(text_input));
466         }
467     }
468
469     if (text_input != selected_input) {
470         text_input->cursor_index = 0;
471         text_input->text_start_index = 0;
472     }
473
474     tekChainThrow(tekGuiTextInputRecreateText(text_input));
475
476     return SUCCESS;
477 }
478
479 /**
480 * Delete a text input gui element, freeing any allocated
481 memory.
482 */
483 void tekGuiDeleteTextInput(TekGuiTextInput* text_input) {
484     tekGuiDeleteButton(&text_input->button); // delete internal
485     button
486     vectorDelete(&text_input->text); // delete written text.
487 }
```

tekgui/text_input.h

```

1 #pragma once
2
3 #include "button.h"
4 #include "../core/vector.h"
5 #include "../tekgl/text.h"
6 #include <cglm/vec4.h>
7
8 struct TekGuiTextInput;
9
10 typedef exception(*TekGuiTextInputCallback)(struct
11 TekGuiTextInput* text_input, const char* text, uint len_text);
12
13 typedef struct TekGuiTextInput {
14     TekGuiButton button;
15     vec4 background_colour;
16     vec4 border_colour;
```

```

16     TekText tek_text;
17     TekText tek_text_cursor;
18     Vector text;
19     uint text_height;
20     vec4 text_colour;
21     uint text_start_index;
22     int text_max_length;
23     uint cursor_index;
24     uint border_width;
25     uint mesh_index;
26     TekGuiTextInputCallback callback;
27     void* data;
28 } TekGuiTextInput;
29
30 exception tekGuiCreateTextInput(TekGuiTextInput* text_input);
31 exception tekGuiDrawTextInput(const TekGuiTextInput*
text_input);
32 exception tekGuiSetTextInputPosition(TekGuiTextInput*
text_input, int x_pos, int y_pos);
33 exception tekGuiSetTextInputSize(TekGuiTextInput* text_input,
uint width, uint height);
34 exception tekGuiSetTextInputText(TekGuiTextInput* text_input,
const char* text);
35 void tekGuiDeleteTextInput(TekGuiTextInput* text_input);

```

tekgui/window.c

```

1 #include "tekgui.h"
2 #include "window.h"
3
4 #include <stdio.h>
5 #include <string.h>
6 #include <cglm/vec2.h>
7 #include "glad/glad.h"
8 #include <GLFW/glfw3.h>
9
10 #include "../tekgl/manager.h"
11 #include "../core/vector.h"
12 #include "../core/yml.h"
13 #include "../tekgl/shader.h"
14 #include "../core/list.h"
15 #include "box_manager.h"
16
17 #define NOT_INITIALISED 0

```

```

18 #define INITIALISED 1
19 #define DE_INITIALISED 2
20
21 /**
22  * Return the largest of two numbers
23  * @param a First number
24  * @param b Second number
25  * @return The largest number
26 */
27 #define MAX(a, b) (a > b) ? a : b;
28
29 static List window_ptr_list;
30
31 static flag window_init = NOT_INITIALISED;
32 static flag window_gl_init = NOT_INITIALISED;
33
34 static struct TekGuiWindowDefaults window_defaults;
35 static GLFWcursor* move_cursor;
36
37 /**
38  * Delete callback for window, freeing any supporting data
39 structures.
40 */
41 static void tekGuiWindowDelete() {
42     // delete window list
43     listDelete(&window_ptr_list);
44
45     window_init = DE_INITIALISED;
46     window_gl_init = DE_INITIALISED;
47 }
48 /**
49  * Callback for when opengl loads. Loads window defaults and
50 creates the cursor for when window is moving.
51  * @throws YML_EXCEPTION if could not load defaults
52 */
53 static exception tekGuiWindowGLLoad() {
54     // load defaults
55     tekChainThrow(tekGuiGetWindowDefaults(&window_defaults));
56
57     // create the cursor
58     move_cursor =
59     glfwCreateStandardCursor(GLFW_CROSSHAIR_CURSOR);

```

```

58
59     window_gl_init = INITIALISED;
60     return SUCCESS;
61 }
62
63 /**
64 * Initialise some supporting data structures for the window
code. Loads some callbacks.
65 */
66 tek_init tekGuiWindowInit() {
67     // load callbacks
68     exception tek_exception =
tekAddDeleteFunc(tekGuiWindowDelete);
69     if (tek_exception) return;
70
71     tek_exception = tekAddGLLoadFunc(tekGuiWindowGLLoad);
72     if (tek_exception) return;
73
74     // create window list
75     listCreate(&window_ptr_list);
76
77     window_init = INITIALISED;
78 }
79
80 /**
81 * Get the data for a window box.
82 * @param window The window to get the box data for.
83 * @param box_data The outputted box data.
84 */
85 static void tekGuiGetWindowData(const TekGuiWindow* window,
TekGuiBoxData* box_data) {
86     // minmax = minimum and maximum extent in that axis.
87     // minmax_i = internal extents, excludes border.
88     glm_vec2_copy((vec2){(float)window->x_pos - (float)window-
>border_width, (float)(window->width + window->x_pos + window-
>border_width)}, box_data->minmax_x);
89     glm_vec2_copy((vec2){(float)window->y_pos - (float)window-
>title_width, (float)(window->height + window->y_pos + window-
>border_width)}, box_data->minmax_y);
90     glm_vec2_copy((vec2){(float)window->x_pos, (float)(window-
>x_pos + window->width)}, box_data->minmax_ix);
91     glm_vec2_copy((vec2){(float)window->y_pos, (float)(window-
>y_pos + window->height)}, box_data->minmax_iy);

```

```

92  }
93
94 /**
95  * Create a box mesh for the window and add it to the list of
96 boxes (all stored in one buffer)
97 * @param window The window to create the box for.
98 * @param index The outputted index of the box in the buffer of
99 boxes.
100 * @throws OPENGL_EXCEPTION .
101 */
102 static exception tekGuiWindowAddGLMesh(const TekGuiWindow*
window, uint* index) {
103     // get box data and create box
104     TekGuiBoxData box_data = {};
105     tekGuiGetWindowData(window, &box_data);
106     tekChainThrow(tekGuiCreateBox(&box_data, index));
107     return SUCCESS;
108 }
109 /**
110  * Update the mesh of the background area of the window.
111 * @param window The window to update.
112 * @throws OPENGL_EXCEPTION .
113 */
114 static exception tekGuiWindowUpdateGLMesh(const TekGuiWindow*
window) {
115     // get box data and update the box.
116     TekGuiBoxData box_data = {};
117     tekGuiGetWindowData(window, &box_data);
118     tekChainThrow(tekGuiUpdateBox(&box_data, window-
>mesh_index));
119     return SUCCESS;
120 }
121 /**
122  * Bring a window to the front so it is rendered above other
123 windows.
124 * @param window The window to bring to the front.
125 * @throws LIST_EXCEPTION if could not move window in the
126 window list.
127 * @throws EXCEPTION whatever the select callback of the window
128 threw.
129 */

```

```

127 exception tekGuiBringWindowToFront(const TekGuiWindow* window)
{
128     const ListItem* item = 0;
129     uint index = 0;
130     foreach(item, (&window_ptr_list), {
131         if (item->data == window) break;
132         index++;
133     });
134
135     // change the order of the window pointer list
136     tekChainThrow(listMoveItem(&window_ptr_list, index,
137     window_ptr_list.length - 1));
138
139     // make sure that the window buttons are also brought to
140     // the front.
141     tekChainThrow(tekGuiBringButtonToFront(&window-
142     >title_button));
143
144
145     return SUCCESS;
146 }
147
148 /**
149 * Update the position of the title button hitbox according to
150 * the window position.
151 * @param window The window to update the title button of.
152 */
153 static void tekGuiWindowUpdateButtonHitbox(TekGuiWindow*
154 window) {
155     // dont think i need to cast to uint any more cuz i fixed
156     // the buttons but oh well
157     window->title_button.hitbox_x = (uint)MAX(0, window->x_pos
158     - (int)window_defaults.border_width);
159     window->title_button.hitbox_y = (uint)MAX(0, window->y_pos
160     - (int)window_defaults.title_width);
161 }
162
163 /**
164 * Create the visual title text of a window.
165 * @param window The window to create the title text of.

```

```

161     * @throws FREETYPE_EXCEPTION .
162     * @throws OPENGL_EXCEPTION .
163     */
164 static exception tekGuiWindowCreateTitleText(TekGuiWindow*
window) {
165     // hard coded ratio that looks nice
166     const uint text_size = window->title_width * 4 / 5;
167
168     // get font and create text
169     TekBitmapFont* font;
170     tekChainThrow(tekGuiGetDefaultFont(&font));
171     tekChainThrow(tekCreateText(window->title, text_size, font,
&window->title_text));
172     return SUCCESS;
173 }
174
175 /**
176     * Recreate the title text visually of the window.
177     * @param window The window to recreate the title text of.
178     * @throws OPENGL_EXCEPTION .
179 */
180 static exception tekGuiWindowRecreateTitleText(TekGuiWindow*
window) {
181     // hard coded ratio that looks nice
182     const uint text_size = window->title_width * 4 / 5;
183     tekChainThrow(tekUpdateText(&window->title_text, window-
>title, text_size));
184     return SUCCESS;
185 }
186
187 /**
188     * Callback for whenever the title area of a window is clicked.
Handles dragging the window.
189     * @param button_ptr Pointer to the button to register clicks
to the title area.
190     * @param callback_data The callback data such as which button
was pressed.
191 */
192 static void tekGuiWindowTitleButtonCallback(TekGuiButton*
button_ptr, TekGuiButtonCallbackData callback_data) {
193     // get the window from button data.
194     TekGuiWindow* window = (TekGuiWindow*)button_ptr->data;
195     switch (callback_data.type) {

```

```

196     case TEK_GUI_BUTTON_MOUSE_BUTTON_CALLBACK: // on clicks
197         // if not a left click, ignore it.
198         if (callback_data.data.mouse_button.button != GLFW_MOUSE_BUTTON_LEFT) break;
199
200         // on press, begin dragging the window
201         if (callback_data.data.mouse_button.action == GLFW_PRESS) {
202             window->being_dragged = 1;
203             // need a delta, cuz the mouse aint gonna be at the
204             // top left of the window always.
205             window->x_delta = window->x_pos -
206             (int)callback_data.mouse_x;
207             window->y_delta = window->y_pos -
208             (int)callback_data.mouse_y;
209             tekSetCursor(CROSSHAIR_CURSOR); // make cursor look
210             like its moving the window
211             tekGuiBringWindowToFront(window);
212             // on release, stop dragging the window
213             } else if (callback_data.data.mouse_button.action == GLFW_RELEASE) {
214                 window->being_dragged = 0;
215                 tekSetCursor(DEFAULT_CURSOR); // reset the cursor
216             }
217             break;
218
219         case TEK_GUI_BUTTON_MOUSE_LEAVE_CALLBACK: // if mouse goes
220             out of the border, reset
221             // kinda a bodge, but this is the best solution cuz my
222             // buttons dont register events outside of their region
223             window->being_dragged = 0;
224             tekSetCursor(DEFAULT_CURSOR);
225             break;
226
227         case TEK_GUI_BUTTON_MOUSE_TOUCHING_CALLBACK: // while mouse
228             is moving
229             if (window->being_dragged) { // if held while moving,
230             move the window to the mouse
231                 window->being_dragged = 1;
232                 window->x_pos = (int)callback_data.mouse_x +
233                 window->x_delta;
234                 window->y_pos = (int)callback_data.mouse_y +
235                 window->y_delta;
236                 // moving window = needs to be redrawn
237                 tekGuiWindowUpdateButtonHitbox(window);

```

```

226         tekGuiWindowUpdateGLMesh(window);
227     }
228     break;
229 default:
230     break;
231 }
232 }
233
234 /**
235 * Update the string buffer that contains the window title,
236 this will allocate or reallocate a buffer as needed, and copy in the
237 title string.
238 * @param window The window to update the title buffer of.
239 * @param title The new title to put in the buffer.
240 * @throws MEMORY_EXCEPTION if malloc() fails.
241 */
242 static exception tekGuiSetTitleBuffer(TekGuiWindow*
window, const char* title) {
243     // free title if exists
244     if (window->title)
245         free(window->title);
246
247     // mallocate buffer
248     const uint len_title = strlen(title) + 1;
249     window->title = (char*)malloc(len_title * sizeof(char));
250     if (!window->title)
251         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for title.");
252
253     // copy title into buffer
254     memcpy(window->title, title, len_title);
255     return SUCCESS;
256 }
257 /**
258 * Set the title of a window in the gui (not the main window).
259 Title is copied, changing original string does not affect the title.
260 * @param window The window to set the title of.
261 * @param title The new title of the window.
262 * @throws MEMORY_EXCEPTION if malloc() fails.
263 */
264 exception tekGuiSetTitle(TekGuiWindow* window, const
char* title) {

```

```

263     // copy title into buffer and redraw title
264     tekChainThrow(tekGuiSetTitleBuffer(window, title));
265     tekChainThrow(tekGuiWindowRecreateTitleText(window));
266     return SUCCESS;
267 }
268
269 /**
270 * Create a new window and add it to the list of all windows.
271 * @param window The new outputted window.
272 * @throws FAILURE if opengl not initialised.
273 * @throws OPENGL_EXCEPTION .
274 */
275 exception tekGuiCreateWindow(TekGuiWindow* window) {
276     // check if opengl initialised + window intitialised
277     if (!window_init) tekThrow(FAILURE, "Attempted to run
function before initialised.");
278
279     // assign all default values from defaults.yml
280     window->type = WINDOW_TYPE_EMPTY;
281     window->visible = 1;
282     window->data = NULL;
283     window->x_pos = (int)window_defaults.x_pos;
284     window->y_pos = (int)window_defaults.y_pos;
285     window->width = window_defaults.width;
286     window->height = window_defaults.height;
287     window->title_width = window_defaults.title_width;
288     window->border_width = window_defaults.border_width;
289     glm_vec4_copy(window_defaults.background_colour, window-
>background_colour);
290     glm_vec4_copy(window_defaults.border_colour, window-
>border_colour);
291     glm_vec4_copy(window_defaults.title_colour, window-
>title_colour);
292
293     // create the window title char buffer and the
corresponding text mesh.
294     window->title = NULL;
295     tekChainThrow(tekGuiSetTitleBuffer(window,
window_defaults.title));
296     tekChainThrow(tekGuiWindowCreateTitleText(window));
297
298     // create button to detect when title bar clicked / dragged
299     tekChainThrow(tekGuiCreateButton(&window->title_button));

```

```

300     tekGuiWindowUpdateButtonHitbox(window);
301     window->title_button.hitbox_width = window_defaults.width +
2 * window_defaults.border_width;
302     window->title_button.hitbox_height =
window_defaults.title_width;
303     window->title_button.data = (void*)window;
304     window->title_button.callback =
tekGuiWindowTitleButtonCallback;
305
306     // add background mesh to list
307     tekChainThrow(tekGuiWindowAddGLMesh(window, &window-
>mesh_index));
308
309     window->being_dragged = 0;
310     window->x_delta = 0;
311     window->y_delta = 0;
312
313     tekChainThrow(listAddItem(&window_ptr_list, window));
314
315     window->draw_callback = NULL;
316
317     return SUCCESS;
318 }
319
320 /**
321 * Draw a single window, and call any sub draw calls of the
322 * window.
323 * @param window The window to draw.
324 * @throws OPENGL_EXCEPTION .
325 */
326 exception tekGuiDrawWindow(const TekGuiWindow* window) {
327     if (!window->visible) return SUCCESS;
328
329     // draw background box
330     tekChainThrow(tekGuiDrawBox(window->mesh_index, window-
>background_colour, window->border_colour));
331
332     // draw title
333     const float x = (float)(window->x_pos + (int)(window->width
/ 2)) - window->title_text.width / 2.0f;
334     const float y = (float)(window->y_pos - (int)window-
>title_width);

```

```

334     tekChainThrow(tekDrawColouredText(&window->title_text, x,
335     y, window->title_colour));
336
337     // callback for additional drawing the window would want to
338     do
339         if (window->draw_callback)
340             tekChainThrow(window->draw_callback(window));
341
342
343 /**
344 * Draw all the windows that have been created.
345 * @throws OPENGL_EXCEPTION .
346 */
347 exception tekGuiDrawAllWindows() {
348     // loop through windows and draw each one
349     const ListItem* item = 0;
350     foreach(item, (&window_ptr_list), {
351         const TekGuiWindow* window = (const TekGuiWindow*)item-
352             >data;
353         tekChainThrow(tekGuiDrawWindow(window));
354     });
355     return SUCCESS;
356 }
357 /**
358 * Set the position of a window in pixels.
359 * @param window The window to set the position of.
360 * @param x_pos The new x position of the window.
361 * @param y_pos The new y position of the window.
362 */
363 void tekGuiSetWindowPosition(TekGuiWindow* window, const int
x_pos, const int y_pos) {
364     // update position and redraw mesh
365     window->x_pos = x_pos;
366     window->y_pos = y_pos;
367     tekGuiWindowUpdateButtonHitbox(window);
368     tekGuiWindowUpdateGLMesh(window);
369 }
370
371 /**
372 * Set the size of a window in pixels.

```

```

373     * @param window The window to set the size of.
374     * @param width The new width in pixels.
375     * @param height The new height in pixels.
376     */
377 void tekGuiSetWindowSize(TekGuiWindow* window, const uint
width, const uint height) {
378     // update size and redraw mesh and update button
379     window->width = width;
380     window->height = height;
381     tekGuiWindowUpdateButtonHitbox(window);
382     tekGuiWindowUpdateGLMesh(window);
383 }
384
385 /**
386     * Set the background colour of a window.
387     * @param window The window to set the background colour of.
388     * @param colour The colour to set the background as.
389     */
390 void tekGuiSetWindowBackgroundColour(TekGuiWindow* window, vec4
colour) {
391     // set colour and update mesh
392     glm_vec4_copy(colour, window->background_colour);
393     tekGuiWindowUpdateGLMesh(window);
394 }
395
396 /**
397     * Set the colour of the border of a window.
398     * @param window The window to set the border colour of.
399     * @param colour The colour to set the window border.
400     */
401 void tekGuiSetWindowBorderColour(TekGuiWindow* window, vec4
colour) {
402     // set colour and update mesh.
403     glm_vec4_copy(colour, window->border_colour);
404     tekGuiWindowUpdateGLMesh(window);
405 }
406
407 /**
408     * Delete a window, freeing any allocated memory.
409     * @param window The window to delete.
410     */
411 void tekGuiDeleteWindow(TekGuiWindow* window) {
412     // free everything we allocated

```

```
413     tekGuiDeleteButton(&window->title_button);
414     tekDeleteText(&window->title_text);
415     free(window->title);
416 }
```

tekgui/window.h

```
1  #pragma once
2
3  #include "../tekgl.h"
4  #include "../core/exception.h"
5  #include <cglm/vec4.h>
6  #include "../tekgl/mesh.h"
7  #include "button.h"
8  #include "../tekgl/text.h"
9
10 #define WINDOW_TYPE_EMPTY 0
11
12 struct TekGuiWindow;
13
14 typedef exception(*TekGuiWindowDrawCallback)(struct
15 TekGuiWindow* window);
16
17 typedef struct TekGuiWindow {
18     flag type;
19     flag visible;
20     void* data;
21     int x_pos;
22     int y_pos;
23     uint width;
24     uint height;
25     uint title_width;
26     uint border_width;
27     vec4 background_colour;
28     vec4 border_colour;
29     vec4 title_colour;
30     uint mesh_index;
31     char* title;
32     TekText title_text;
33     TekGuiButton title_button;
34     flag being_dragged;
35     int x_delta;
```

```

36     int y_delta;
37     TekGuiWindowDrawCallback draw_callback;
38     TekGuiWindowSelectCallback select_callback;
39 } TekGuiWindow;
40
41 exception tekGuiBringWindowToFront(const TekGuiWindow* window);
42 exception tekGuiCreateWindow(TekGuiWindow* window);
43 exception tekGuiCreateWindowSAFE(TekGuiWindow* window);
44 void tekGuiSetWindowPosition(TekGuiWindow* window, int x_pos,
int y_pos);
45 void tekGuiSetWindowSize(TekGuiWindow* window, uint width, uint
height);
46 exception tekGuiSetTitle(TekGuiWindow* window, const char*
title);
47 void tekGuiSetWindowBackgroundColour(TekGuiWindow* window, vec4
colour);
48 void tekGuiSetWindowBorderColour(TekGuiWindow* window, vec4
colour);
49 void tekGuiDeleteWindow(TekGuiWindow* window);
50 exception tekGuiDrawWindow(const TekGuiWindow* window);
51 exception tekGuiDrawAllWindows();

```

tekphys/body.c

```

1 #include "body.h"
2
3 #include "geometry.h"
4 #include <cglm/vec3.h>
5 #include <math.h>
6 #include <stdio.h>
7 #include "../tekgl/manager.h"
8 #include "collider.h"
9
10 /// Struct containing the volume and centre of a tetrahedron
11 struct TetrahedronData {
12     float volume;
13     vec3 centroid;
14 };
15
16 /**
17 * @brief Calculate the volume, centre of mass and inverse
18 *        inertia tensor of an object.
19 * @param body The body to calculate properties of.
20 * @throws MEMORY_EXCEPTION if malloc() fails.

```

```

20  /*
21  static exception tekCalculateBodyProperties(TekBody* body) {
22      // create an array to cache some data about tetrahedra
23      // avoids recalculation later on.
24      const uint len_tetrahedron_data = body->num_indices / 3;
25      struct TetrahedronData* tetrahedron_data = (struct
TetrahedronData*)malloc(len_tetrahedron_data * sizeof(struct
TetrahedronData));
26      if (!tetrahedron_data)
27          tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
to cache tetrahedron data.");
28
29      // inspiration for this algorithm from: http://number-
none.com/blow/inertia/body_i.html
30      // the premise of this algorithm is to iterate over each
triangle in the mesh
31      // an arbitrary point is picked (here i chose 0, 0, 0 for
ease) and each triangle forms a tetrahedron with this point
32      // then you can calculate the volume, centre of mass and
inertia tensor for each tetrahedron
33      // this can be used to find the volume, centre of mass and
inertia tensor for the whole mesh.
34
35      vec3 origin = { 0.0f, 0.0f, 0.0f };
36      vec3 weighted_sum = { 0.0f, 0.0f, 0.0f };
37      float volume = 0.0f;
38
39      // first loop to find the centre of mass
40      for (uint i = 0; i < body->num_indices; i += 3) {
41          // calculate signed volume per tetrahedron
42          const float tetra_volume = tetrahedronSignedVolume(
43              origin,
44              body->vertices[body->indices[i]],
45              body->vertices[body->indices[i + 1]],
46              body->vertices[body->indices[i + 2]]
47          );
48
49          // some will have negative or positive volume
50          // causes overlapping tetrahedra to "cancel out" and
give the correct volume
51          volume += tetra_volume;
52

```

```

53         // for all simplexes, centre of mass (CoM) is average
position of vertices.
54         vec3 tetra_centroid;
55         sumVec3(tetra_centroid, origin, body->vertices[body-
>indices[i]], body->vertices[body->indices[i + 1]], body-
>vertices[body->indices[i + 2]]);
56         glm_vec3_scale(tetra_centroid, 0.25f, tetra_centroid);
57
58         // for whole object, CoM is the weighted average of
tetrahedron CoMs
59         glm_vec3_muladds(tetra_centroid, tetra_volume,
weighted_sum);
60
61         // cache the volume and centroid for next loop
62         const uint index = i / 3;
63         tetrahedron_data[index].volume = tetra_volume;
64         glm_vec3_copy(tetra_centroid,
tetrahedron_data[index].centroid);
65     }
66
67     // divide the weighted sum by total volume to get the
centre of mass
68     glm_vec3_scale(weighted_sum, 1.0f / volume, body-
>centre_of_mass);
69
70     // in opengl, anticlockwise faces are outwards
71     // in maths, clockwise faces are outwards
72     // this leads to meshes appearing to be "inside out" and
having negative volume
73     // so we should take absolute value of volume
74     // as long as face orientation is consistent however, this
shouldn't affect much
75     body->volume = fabsf(volume);
76     body->density = body->mass / body->volume;
77
78     // prepare the inertia tensor, as we will be adding to it
79     mat3 inertia_tensor;
80     glm_mat3_zero(inertia_tensor);
81
82     // second iteration
83     for (uint i = 0; i < body->num_indices; i += 3) {
84         // retrieve stored information about this tetrahedron
85         const uint index = i / 3;

```

```

86         const float mass = fabsf(tetrahedron_data[index].volume
* body->density);
87
88         // calculate inertia tensor for this tetrahedron
89         mat3 tetrahedron_inertia_tensor;
90         tetrahedronInertiaTensor(origin, body->vertices[body-
>indices[i]], body->vertices[body->indices[i + 1]], body-
>vertices[body->indices[i + 2]], mass, tetrahedron_inertia_tensor);
91
92         // translation vector from CoM of object, to CoM of
tetrahedron
93         vec3 translate;
94         glm_vec3_sub(tetrahedron_data[index].centroid, body-
>centre_of_mass, translate);
95
96         // translate inertia tensor using parallel axis theorem
97         // centre the tensor around the body's centre of mass
98         translateInertiaTensor(tetrahedron_inertia_tensor,
mass, translate);
99
100        // sum the translated tensor with the body tensor, to
find final inertia tensor.
101        mat3Add(inertia_tensor, tetrahedron_inertia_tensor,
inertia_tensor);
102    }
103
104    // store inverse inertia tensor, as this is more useful to
us.
105    glm_mat3_inv(inertia_tensor, body->inverse_inertia_tensor);
106
107    free(tetrahedron_data);
108    return SUCCESS;
109 }
110
111 /**
112 * Update the transformation matrix of a body based on its
current position and rotation.
113 * @param body The body to update.
114 */
115 static void tekBodyUpdateTransform(TekBody* body) {
116     // transform matrix is a mixture of translation and
rotation
117     mat4 translation;

```

```

118     glm_translate_make(translation, body->position);
119     mat4 rotation;
120     glm_quat_mat4(body->rotation, rotation);
121
122     // rotate first, then translate (but matrices work
backwards :D)
123     // cuz rotation only rotates around the origin
124     glm_mat4_mul(translation, rotation, body->transform);
125 }
126
127 /**
128 * @brief Create an instance of a body given an empty TekBody
struct.
129 * @note Will calculate properties of the body given the mesh
data, so could take time for larger objects. Will also allocate
memory to store vertices.
130 * @param mesh_filename The mesh file to use when creating the
body.
131 * @param mass The mass of the object
132 * @param friction Coefficient of friction for the body
133 * @param restitution Coefficient of restitution for the body
134 * @param position The position (x, y, z) of the body's center
of mass
135 * @param rotation The rotation quaternion of the body.
136 * @param scale The scaling applied to the object
137 * @param body A pointer to a struct to contain the new body.
138 * @throws MEMORY_EXCEPTION if malloc() fails.
139 * @throws FAILURE if file is malformed
140 */
141 exception tekCreateBody(const char* mesh_filename, const float
mass, const float friction, const float restitution, vec3 position,
vec4 rotation, vec3 scale, TekBody* body) {
142     // some variables used throughout
143     float* vertex_array = 0;
144     uint* index_array = 0;
145     int* layout_array = 0;
146     uint len_vertex_array = 0, len_index_array = 0,
len_layout_array = 0;
147     uint position_layout_index = 0;
148
149     // read mesh data from the file. read into arrays of
vertices, indices and layout

```

```

150     tekChainThrow(tekReadMeshArrays(mesh_filename,
151     &vertex_array, &len_vertex_array, &index_array, &len_index_array,
152     &layout_array, &len_layout_array, &position_layout_index));
153
154     int vertex_size = 0;
155     int position_index = 0;
156
157     // check position layout index. should be a 3 vector
158     for (uint i = 0; i < len_layout_array; i++) {
159         if (position_layout_index == i) {
160             if (layout_array[i] != 3) tekThrow(FAILURE,
161             "Position data must be 3 floats.");
162             position_index = vertex_size;
163         }
164         vertex_size += layout_array[i];
165     }
166
167     // no longer needed, only useful for rendering
168     free(layout_array);
169
170     // allocate memory for body vertices and copy them in
171     const uint num_vertices = len_vertex_array / vertex_size;
172     body->vertices = (vec3*)malloc(num_vertices *
173     sizeof(vec3));
174     if (!body->vertices) tekThrow(MEMORY_EXCEPTION, "Failed to
175     allocate memory for vertices.");
176     for (uint i = 0; i < num_vertices; i++) {
177         for (uint j = 0; j < 3; j++) {
178             body->vertices[i][j] = vertex_array[i * vertex_size
179             + position_index + j];
180         }
181     }
182
183     // copy other values into the body
184     body->num_vertices = num_vertices;
185     body->indices = index_array;
186     body->num_indices = len_index_array;
187     body->mass = mass;
188     body->friction = friction;
189     body->restitution = restitution;
190     glm_vec3_copy(position, body->position);
191     glm_vec4_copy(rotation, body->rotation);
192     glm_vec3_copy(scale, body->scale);

```

```

187
188     // calculate properties - centre of mass, inertia tensor
189     tekChainThrow(tekCalculateBodyProperties(body));
190
191     // create the collider structure
192     tekChainThrow(tekCreateCollider(body, &body->collider));
193
194     // create transformation matrix
195     tekBodyUpdateTransform(body);
196
197     return SUCCESS;
198 }
199
200 /**
201 * @brief Simulate the effect of a certain amount of time
202 passing on the body's position and rotation.
203 * @note Simulate linearly, e.g. with constant acceleration
204 between the two points in time.
205 * @param body The body to advance forward in time.
206 * @param delta_time The length of time to advance by.
207 * @param gravity The downwards acceleration due to gravity.
208 */
209 void tekBodyAdvanceTime(TekBody* body, const float delta_time,
210 const float gravity) {
211     if (body->immovable) {
212         tekBodyUpdateTransform(body);
213         return;
214     }
215
216     vec3 delta_position = { 0.0f, 0.0f, 0.0f };
217     body->velocity[1] -= gravity * delta_time;
218     glm_vec3_scale(body->velocity, delta_time, delta_position);
219     glm_vec3_add(body->position, delta_position, body-
220 >position);
221
222     // calculating change in angle due to angular velocity is a
223 LOT more complex than anticipated
224     // so here is a lot of comments so hopefully I can
225 understand later on
226
227     // the magnitude of angular velocity = angular speed (in
228 radians per second).
229     // so multiplying by delta time gives us a change in angle.

```

```

223     const float delta_angle = glm_vec3_norm(body-
>angular_velocity) * delta_time;
224
225     // for small angles, floating point precision may start to
become a problem
226     // therefore, safe to skip these small angles as rotation
change may be smaller than floating point precision anyway
227     if (delta_angle > 1e-5f) {
228         // normalise angular velocity vector to get the axis of
rotation as a unit vector
229         // e.g. { 0.0, 1.0, 0.0 } would mean rotation around
the Y axis
230         vec3 axis;
231         glm_vec3_normalize_to(body->angular_velocity, axis);
232
233         // calculate a quaternion that represents the change in
rotation
234         // e.g. create a rotation around "axis" of size "angle"
235         vec4 delta_quat;
236         glm_quatv(delta_quat, delta_angle, axis);
237
238         // apply the rotation by multiplying rotation change by
current rotation
239         vec4 result;
240         glm_quat_mul(delta_quat, body->rotation, result);
241
242         // normalise the quaternion, as this rotation causes
slight errors that need to be corrected
243         glm_quat_normalize(result);
244
245         glm_quat_copy(result, body->rotation);
246     }
247
248     tekBodyUpdateTransform(body);
249 }
250
251 /**
252 * @brief Apply an impulse (change in momentum) to a body.
253 * @note The point of application is in world coordinates, not
relative to the body.
254 * @param body The body to apply the impulse to.
255 * @param point_of_application The point in space where the
impulse is applied.

```

```

256     * @param impulse The size of the impulse to apply.
257     * @param delta_time The duration of time for which this
258     * impulse takes place. Should be the same as the physics time step
259     * typically.
260     */
261     void tekBodyApplyImpulse(TekBody* body, vec3
262     point_of_application, vec3 impulse, const float delta_time) {
263         // impulse = mass * Δvelocity
264         glm_vec3_muladds(impulse, 1.0f / body->mass, body-
265         >velocity);
266
267         // calculating force:
268         // force = impulse / time
269         vec3 force;
270         glm_vec3_scale(impulse, 1.0f / delta_time, force);
271
272         // calculating torque:
273         // τ = r × F
274         vec3 displacement;
275         glm_vec3_sub(point_of_application, body->centre_of_mass,
276         displacement);
277         vec3 torque;
278         glm_vec3_cross(displacement, force, torque);
279
280         // calculating angular acceleration:
281         // α = I⁻¹τ
282         vec3 angular_acceleration;
283         glm_mat3_mulv(body->inverse_inertia_tensor, torque,
284         angular_acceleration);
285
286         // add change in angular velocity (angular acceleration *
287         // Δtime = Δangular velocity)
288         glm_vec3_muladds(angular_acceleration, delta_time, body-
289         >angular_velocity);
290     }
291
292     /**
293      * Update the mass of a body. Don't set the mass directly
294      * because the mass affects the density, inertia tensor and other
295      * properties that need to be updated.
296      * @param body The body to have its mass changed.
297      * @param mass The new mass of the body.
298      * @throws MEMORY_EXCEPTION if malloc() fails

```

```

289     */
290 exception tekBodySetMass(TekBody* body, const float mass) {
291     body->mass = mass; // kiss my mass
292     // recalculate some properties that are based on mass
293     tekChainThrow(tekCalculateBodyProperties(body));
294     return SUCCESS;
295 }
296
297 /**
298 * @brief Delete a TekBody by freeing the vertices that were
299 allocated.
300 * @param body The body to delete.
301 */
302 void tekDeleteBody(const TekBody* body) {
303     // delete the collider before freeing so we dont lose the
304     // pointer
305     tekDeleteCollider(&body->collider);
306     free(body->vertices);
307 }
```

tekphys/body.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../tekgl/mesh.h"
5 #include "../tekgl/material.h"
6 #include "../core/exception.h"
7 #include "../core/vector.h"
8
9 #include <cglm/vec3.h>
10 #include <cglm/vec4.h>
11 #include <cglm/quat.h>
12
13 struct TekColliderNode;
14 typedef struct TekColliderNode* TekCollider;
15
16 typedef struct TekBody {
17     vec3* vertices;
18     uint num_vertices;
19     uint* indices;
20     uint num_indices;
21     float mass;
22     float density;
```

```

23     float volume;
24     float restitution;
25     float friction;
26     vec3 centre_of_mass;
27     vec3 position;
28     vec3 velocity;
29     vec4 rotation;
30     vec3 angular_velocity; // direction = axis of rotation,
magnitude = speed of rotation (radians/second)
31     vec3 scale;
32     mat3 inverse_inertia_tensor;
33     mat4 transform;
34     TekCollider collider;
35     int immovable;
36 } TekBody;
37
38 typedef struct TekBodySnapshot {
39     float mass;
40     float friction;
41     float restitution;
42     vec3 position;
43     vec4 rotation;
44     vec3 velocity;
45     vec3 angular_velocity;
46     int immovable;
47     char* model;
48     char* material;
49 } TekBodySnapshot;
50
51 exception tekCreateBody(const char* mesh_filename, float mass,
float friction, float restitution, vec3 position, vec4 rotation,
vec3 scale, TekBody* body);
52 void tekBodyAdvanceTime(TekBody* body, float delta_time, float
gravity);
53 void tekDeleteBody(const TekBody* body);
54 void tekBodyApplyImpulse(TekBody* body, vec3
point_of_application, vec3 impulse, float delta_time);
55 exception tekBodySetMass(TekBody* body, float mass);
56 exception tekBodyGetContactPoints(const TekBody* body_a, const
TekBody* body_b, Vector* contact_points);

```

tekphys/collider.c

```
1 #include "collider.h"
```

```

2
3 #include <stdio.h>
4 #include <string.h>
5
6 #include <cglm/vec3-ext.h>
7 #include <cglm/mat3.h>
8
9 #include "geometry.h"
10 #include "body.h"
11 #include "engine.h"
12 #include "../tekgl/manager.h"
13
14 #define LEFT 0
15 #define RIGHT 1
16
17 /**
18 * Single-precision symmetric matrix compute eigenvalues (and
optionally eigenvectors)\.
19 * @param[in] jobz The type of operation, 'N' for only
eigenvalues, 'V' for both.
20 * @param[in] uplo The triangle (of the matrix) to keep, 'U'
for upper, 'L' for lower.
21 * @param[in] n The order of the matrix (n x n).
22 * @param[in/out] a Used to input the matrix, and to output the
eigenvectors.
23 * @param[in] lda The leading dimension of the matrix.
24 * @param[out] w The outputted array of eigenvalues.
25 * @param[in] work The workspace array.
26 * @param[in] lwork The size of the workspace array
(recommended size is max(1, 3n-1)).
27 * @param[out] info The output of the function, 0 on success,
<0 on error, >0 on no solution.
28 * @note External function, interfaces Fortran subprocedure in
LAPACK.
29 */
30 extern void ssyev_(char* jobz, char* uplo, int* n, float* a,
int* lda, float* w, float* work, float* lwork, int* info);
31
32 /**
33 * Single-precision compute solution to a real system of linear
equations.
34 * @param n[in] The number of linear equations.
35 * @param nrhs[in] The number of right hand sides.

```

```

36  * @param a[in,out] The NxN coefficient matrix A.
37  * @param lda[in] The leading dimension of the array A.
38  * @param ipiv[out] The pivot indices that define the
permutation matrix P.
39  * @param b[in,out] The NxNRHS matrix B. Also outputs the
NxNRHS solution matrix X.
40  * @param ldb[in] The leading dimension of the array B.
41  * @param info[out] The output of the function, 0 on success,
<0 on error, >0 on no solution.
42  * @note External function, interfaces Fortran subprocedure in
LAPACK.
43  */
44 extern void sgesv_(int* n, int* nrhs, float* a, int* lda, int*
ipiv, float* b, int* ldb, int* info);
45 /**
46 /**
47 * Compute the eigenvectors and eigenvalues of a symmetric
matrix.
48 * @param[in] matrix The matrix to compute.
49 * @param[in/out] eigenvectors[3] An empty but already
allocated array of 3*sizeof(vec3) that will store the eigenvectors.
50 * @param[in/out] eigenvalues[3] An empty but already allocated
array of 3*sizeof(float) that will store the eigenvalues.
51 * @throws FAILURE if there is no solution or a solution could
not be found in a reasonable time.
52 * @note Uses LAPACK / ssyev_(...) internally.
53 */
54 static exception symmetricMatrixCalculateEigenvectors(mat3
matrix, vec3 eigenvectors[3], float eigenvalues[3]) {
55     // create copy of matrix as LAPACKE will destroy the input.
56     mat3 matrix_copy;
57     glm_mat3_copy(matrix, matrix_copy);
58
59     char jobz = 'V'; // compute both eigenvalues and
eigenvectors.
60     char uplo = 'U'; // use the upper triangle of matrix.
61     int n = 3;        // matrix order
62     float w[3];      // eigenvalues storage
63     float work[8];   // workspace buffer
64     int lwork = 8;    // size of aforementioned buffer
65     int info = 0;     // the output
66
67     // call Fortran function...

```

```

68     // took 4 hours of messing with cmakelists.txt to make this
work.
69     ssyev_(&jobz, &uplo, &n, &matrix_copy, &n, &w, &work,
&lwork, &info);
70     if (info > 0)
71         tekThrow(FAILURE, "Failed to calculate eigenvectors -
failed to converge.");
72     if (info < 0)
73         tekThrow(FAILURE, "Failed to calculate eigenvectors -
illegal value");
74
75     // now, copy eigenvalues into output array.
76     for (uint i = 0; i < 3; i++) {
77         glm_vec3_copy(matrix_copy[i], eigenvectors[i]);
78         eigenvalues[i] = w[i];
79     }
80     return SUCCESS;
81 }
82
83 /**
84 * Solve a simultaneous equation with 3 unknowns and 3 vector
equations.
85 * @param lhs_vectors[3] The three vectors of coefficients for
xs, ys and zs
86 * @param rhs_vector The required result.
87 * @param result The values of x y and z.
88 * @throws FAILURE if there is no solution.
89 */
90 static exception solveSimultaneous3Vec3(vec3 lhs_vectors[3],
vec3 rhs_vector, vec3 result) {
91     // copy lhs and rhs vectors into a buffer so it doesn't get
overwritten by LAPACK
92     vec3 lhs_buffer[3];
93     memcpy(lhs_buffer, lhs_vectors, 3 * sizeof(vec3));
94     vec3 rhs_buffer;
95     glm_vec3_copy(rhs_vector, rhs_buffer);
96
97     int n = 3; // 3 lhs vectors
98     int nrhs = 1; // 1 rhs vector
99     int lda = 3; // 3 components in lhs vectors
100    int ldb = 3; // 3 components in rhs vector
101    int info; // return value
102    int ipiv[3]; // ???

```

```

103
104     sgesv_(&n, &nrhs, lhs_buffer, &lida, &ipiv, rhs_buffer,
105     &ldb, &info);
106     if (info)
107         tekThrow(FAILURE, "Failed to solve simultaneous
equation system.");
108     // result should now be stored in the rhs buffer, return
the answer
109     glm_vec3_copy(rhs_buffer, result);
110
111     return SUCCESS;
112 }
113
114 /**
115 * Calculate the area of a triangle given three points in
space.
116 * @param point_a The first point of the triangle.
117 * @param point_b The second point of the triangle.
118 * @param point_c The third point of the triangle.
119 * @return The area of the triangle formed by those three
points.
120 */
121 static float tekCalculateTriangleArea(vec3 point_a, vec3
point_b, vec3 point_c) {
122     // area = 0.5 * | (b - a) x (c - a) |
123     // magnitude of cross product = area of parallelogram
between two vectors.
124     // area of triangle = half area of parallelogram.
125     vec3 b_a, c_a;
126     glm_vec3_sub(point_b, point_a, b_a);
127     glm_vec3_sub(point_c, point_a, c_a);
128     vec3 cross;
129     glm_vec3_cross(b_a, c_a, cross);
130     return 0.5f * glm_vec3_norm(cross);
131 }
132
133 /**
134 * Ensure that an index is allowed, throw error if not.
135 */
136 #define tekCheckIndex(index) \
137 if (index >= num_vertices) \
138 tekThrowThen(FAILURE, "Vertex does not exist.", { \

```

```

139     vectorDelete(triangles); \
140  }); \
141
142 /**
143  * Fill a Vector with the data for all triangles of a mesh.  

This means the vertices and area of each triangle.
144  * @param vertices An array of vertices, use body.vertices.
145  * @param num_vertices The number of vertices, use  

body.num_vertices.
146  * @param indices An array of indices, use body.indices.
147  * @param num_indices The number of indices, use  

body.num_indices.
148  * @param triangles An existing Vector struct, will have  

vectorCreate(...) called, so you must call vectorDelete(...) after  

finishing with the Vector.
149  * @note 'triangles' is freed if the function fails.
150  * @throws MEMORY_EXCEPTION if malloc() fails.
151 */
152 static exception tekGenerateTriangleArray(const vec3* vertices,
const uint num_vertices, const uint* indices, const uint
num_indices, Vector* triangles) {
153     // 1 triangle : 3 vertices, so triangle list is a third of  

the size of vertices array.
154     tekChainThrow(vectorCreate(num_indices / 3, sizeof(struct
Triangle), triangles));
155
156     // temp struct to fill with triangle data to copy into the  

vector.
157     struct Triangle triangle = {};
158     for (uint i = 0; i < num_indices; i += 3) {
159         // copy each vertex into the triangle struct.
160         tekCheckIndex(indices[i]);
161         glm_vec3_copy(vertices[indices[i]],
triangle.vertices[0]);
162         tekCheckIndex(indices[i + 1]);
163         glm_vec3_copy(vertices[indices[i + 1]],
triangle.vertices[1]);
164         tekCheckIndex(indices[i + 2]);
165         glm_vec3_copy(vertices[indices[i + 2]],
triangle.vertices[2]);
166

```

```

167         triangle.area =
tekCalculateTriangleArea(triangle.vertices[0], triangle.vertices[1],
triangle.vertices[2]);
168
169         // centroid = (sum of vertices) / 3
170         vec3 centroid;
171         glm_vec3_zero(centroid);
172         sumVec3(
173             centroid,
174             triangle.vertices[0],
175             triangle.vertices[1],
176             triangle.vertices[2]
177         );
178         glm_vec3_scale(centroid, 1.0f / 3.0f,
triangle.centroid);
179
180         // add to vector
181         tekChainThrowThen(vectorAddItem(triangles, &triangle),
{
182             vectorDelete(triangles);
183         });
184     }
185     return SUCCESS;
186 }
187
188 /**
189 * Calculate the mean point of a set of triangles interpreted
as a convex hull.
190 * @param[in] triangles A vector containing a list of triangles
(3 vertices, centroid and area).
191 * @param[in] indices An array of indices that determine which
triangles from the triangles vector will be used in the calculation.
192 * @param[in] num_indices The number of indices contained in
the array.
193 * @param[out] mean The calculated mean of the convex hull.
194 */
195 static void tekCalculateConvexHullMean(const Vector* triangles,
const uint* indices, const uint num_indices, vec3 mean) {
196     // https://www.cs.unc.edu/techreports/96-013.pdf
197     // mean of convex hull = (1/(6n)) * SUM((1/area)(a + b +
c))
198     // where n = num triangles
199     // a, b, c = vertices of each triangle.

```

```

200
201     // perform summation
202     glm_vec3_zero(mean);
203     for (uint i = 0; i < num_indices; i++) {
204         vec3 sum;
205         glm_vec3_zero(sum);
206         const struct Triangle* triangle;
207         vectorGetItemPtr(triangles, indices[i], &triangle);
208
209         // (a + b + c)
210         sumVec3(
211             sum,
212             triangle->vertices[0],
213             triangle->vertices[1],
214             triangle->vertices[2]
215         );
216
217         // SUM(1 / area)(a + b + c)
218         glm_vec3_muladds(sum, 1.0f / triangle->area, mean);
219     }
220
221     // divide by 6n to complete calculation
222     const float num_triangles = (float)(6 * triangles->length);
223     glm_vec3_scale(mean, 1.0f / num_triangles, mean);
224 }
225
226 /**
227  * Calculate the covariance matrix of a convex hull. This is a
228  * statistical measure of the spread of "matter" in the rigidbody, or
229  * like the standard deviation in 3 axes.
230  * @param[in] triangles A vector of triangle data (1 triangle =
231  * 3 vertices, centroid and area).
232  * @param[in] indices An array of indices that determine which
233  * triangles are used from the vector.
234  * @param[in] num_indices The length of the indices array.
235  * @param[in] mean The mean point of the convex hull,
236  * calculated using \ref tekCalculateConvexHullMean
237  * @param[out] covariance The calculated covariance matrix of
238  * the convex hull.
239  */
240 static void tekCalculateCovarianceMatrix(const Vector*
triangles, const uint* indices, const uint num_indices, const vec3
mean, mat3 covariance) {

```

```

235     // https://www.cs.unc.edu/techreports/96-013.pdf
236     // Covariance[x, y] = (1/n) * SUM 1->n(area[i] * ((am[x] +
237     // bm[x] + cm[x]) * (am[y] + bm[y] + cm[y]) + am[i]am[j] + bm[i]bm[j] +
238     // cm[i]cm[j]))
239     // a = a - mean, b = b - mean, c = c - mean
240     // a, b, c = vertices of triangle
241     // x, y are row and column of covariance matrix in range
242     // 0..2
243     glm_mat3_zero(covariance);
244     for (uint i = 0; i < num_indices; i++) {
245         struct Triangle* triangle;
246         vectorGetItemPtr(triangles, indices[i], &triangle);
247         for (uint j = 0; j < 3; j++) {
248             for (uint k = 0; k < 3; k++) {
249                 // calculate vertex - mean as variable for
250                 // easier reading
251                 const float pj = triangle->vertices[0][j] -
252                 mean[0];
253                 const float pk = triangle->vertices[0][k] -
254                 mean[0];
255                 const float qj = triangle->vertices[1][j] -
256                 mean[1];
257                 const float qk = triangle->vertices[1][k] -
258                 mean[1];
259                 const float rj = triangle->vertices[2][j] -
260                 mean[2];
261                 const float rk = triangle->vertices[2][k] -
262                 mean[2];
263                 // perform calculation for each element of
264                 // matrix
265                 covariance[k][j] += triangle->area * ((pj + qj
266                 + rj) * (pk + qk + rk) + pj * pk + qj * qk + rj * rk);
267             }
268         }
269     }
270     // divide by number of triangles to complete the
271     // calculation.
272     glm_mat3_scale(covariance, 1.0f / (float)triangles-
273     >length);
274 }
275
276
277 /**

```

```

264  * Find the minimum and maximum point of a set of triangles
projected onto a single axis. Used to find the half extents of an
OBB.
265  * @param[in] triangles A vector containing the triangles of a
mesh.
266  * @param[in] indices An array containing the indices of the
triangles that should be included in the calculation.
267  * @param[in] num_indices The length of the indices array.
268  * @param[in] axis The axis to project along. Should be one of
the eigenvectors of the covariance matrix.
269  * @param[out] min_p A pointer to where the minimum projection
should be stored.
270  * @param[out] max_p A pointer to where the maximum projection
should be stored.
271  */
272 static void tekFindProjections(const Vector* triangles, const
uint* indices, const uint num_indices, vec3 axis, float* min_p,
float* max_p) {
273     // search for maximum and minimum projection along axis
274     float min_proj = INFINITY, max_proj = -INFINITY;
275     for (uint i = 0; i < num_indices; i++) {
276         // read triangle at each index of indices array.
277         const struct Triangle* triangle;
278         vectorGetItemPtr(triangles, indices[i], &triangle);
279         for (uint j = 0; j < 3; j++) {
280             // dot product = find the projection of point along
the axis.
281             const float proj = glm_vec3_dot(triangle-
>vertices[j], axis);
282             if (proj < min_proj) {
283                 min_proj = proj;
284             }
285             if (proj > max_proj) {
286                 max_proj = proj;
287             }
288         }
289     }
290
291     // output minimum and maximum projection
292     *min_p = min_proj;
293     *max_p = max_proj;
294 }
295

```

```

296 /**
297  * Create an OBB based on a set of triangles. This will be the
298  * smallest rectangular prism that contains all points of the
299  * triangles.
300  * @param[in] triangles A vector containing all the triangles
301  * of a mesh.
302  * @param[in] indices An array containing the indices of
303  * triangles which should be used in the calculations.
304  * @param[in] num_indices The number of indices that are in the
305  * array.
306  * @param[out] obb The obb struct that will be filled with the
307  * calculated data.
308  * @throws FAILURE if the calculation of eigenvectors fails.
309  */
310 static exception tekCreateOBB(const Vector* triangles, const
311 uint* indices, const uint num_indices, struct OBB* obb) {
312     // calculate mean and covariance of the set of triangles.
313     // This represents the central tendency and spread of the points
314     vec3 mean;
315     tekCalculateConvexHullMean(triangles, indices, num_indices,
316     mean);
317     mat3 covariance;
318     tekCalculateCovarianceMatrix(triangles, indices,
319     num_indices, mean, covariance);
320
321     // using this, we can find the eigenvectors which represent
322     // three orthogonal vectors of greatest spread.
323     // these eigenvectors are used to construct the OBB by
324     // producing a face perpendicular to each eigenvector.
325     vec3 eigenvectors[3];
326     float eigenvalues[3];
327
328     tekChainThrow(symmetricMatrixCalculateEigenvectors(covariance,
329     eigenvectors, eigenvalues));
330
331     // iterate over each eigenvector to find the centre of the
332     // OBB and the half extents
333     // the half extent is the shortest distance between the
334     // centre and one of the faces (2 * half extent = full extent = height
335     // of OBB in one axis)
336     vec3 centre = {0.0f, 0.0f, 0.0f};
337     for (uint i = 0; i < 3; i++) {
338         float min_proj, max_proj;

```

```

322         tekFindProjections(triangles, indices, num_indices,
eigenvectors[i], &min_proj, &max_proj);
323         const float half_extent = 0.5f * (max_proj - min_proj);
324         const float centre_proj = 0.5f * (max_proj + min_proj);
325         glm_vec3_muladds(eigenvectors[i], centre_proj, centre);
326         glm_vec3_copy(eigenvectors[i], obb->axes[i]);
327         glm_vec3_copy(eigenvectors[i], obb->w_axes[i]);
328         obb->half_extents[i] = half_extent;
329         obb->w_half_extents[i] = half_extent;
330     }
331
332     glm_vec3_copy(centre, obb->centre);
333     glm_vec3_copy(centre, obb->w_centre);
334
335     return SUCCESS;
336 }
337
338 /**
339  * Create a collider node struct given some information. Mostly
just a helper function to create the internal OBB and set some
values rather than copy-pasting a 10 line statement.
340  * @param[in] type The type of collider node, should be either
COLLIDER_NODE or COLLIDER_LEAF.
341  * @param[in] id The id of the collider node (unused for now,
may be needed).
342  * @param[in] triangles A vector of triangles that make up a
mesh (triangle = 3 vertices, centroid and area).
343  * @param[in] indices An array of indices specifying which
triangles should be used from the vector.
344  * @param[in] num_indices The number of indices in the array.
345  * @param[out] collider_node The collider node that was
produced. Should point to an empty pointer that will be set to the
memory address where the collider node was created at.
346  * @throws MEMORY_EXCEPTION if malloc() fails.
347  * @throws FAILURE if there is an error while calculating
eigenvectors.
348 */
349 static exception tekCreateColliderNode(const flag type, const
uint id, const Vector* triangles, uint* indices, const uint
num_indices, TekColliderNode** collider_node) {
350     // allocate memory for collider
351     *collider_node =
(TekColliderNode*)malloc(sizeof(TekColliderNode));

```

```

352     if (!*collider_node)
353         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for collider node.");
354     tekChainThrowThen(tekCreateOBB(triangles, indices,
num_indices, &(*collider_node)->obb), {
355         free(*collider_node);
356         *collider_node = 0;
357     });
358
359     // write data in
360     (*collider_node)->id = id;
361     (*collider_node)->type = type;
362     (*collider_node)->indices = indices;
363     (*collider_node)->num_indices = num_indices;
364
365     // setting left & right pointers to NULL, in case someone
tries to dereference them with junk in there.
366     if (type == COLLIDER_NODE) {
367         (*collider_node)->data.node.left = 0;
368         (*collider_node)->data.node.right = 0;
369     }
370     return SUCCESS;
371 }
372
373 /**
374  * Clean up memory involved in creation of collider.
375  */
376 #define tekColliderCleanup() \
377 { \
378     vectorDelete(&triangles); \
379     vectorDelete(&collider_stack); \
380     tekDeleteCollider(collider); \
381 } \
382
383 /**
384  * Helper function to print out a collider tree.
385  * @param collider The collider structure to print out.
386  * @param indent The indent to print at (set to 0 if calling
this, recursive calls set to indent + 1)
387 */
388 static void tekPrintCollider(TekColliderNode* collider, uint
indent) {
389     if (collider->type == COLLIDER_NODE) {

```

```

390         // using in order traversal, prints bottom left item of
tree as the first line, and branches from there.
391         tekPrintCollider(collider->data.node.left, indent + 1);
392         for (uint i = 0; i < indent; i++) {
393             printf("  ");
394         }
395         printf("%u > (%f %f %f)\n", collider->id,
EXPAND_VEC3(collider->obb.half_extents));
396         tekPrintCollider(collider->data.node.right, indent +
1);
397     } else if (collider->type == COLLIDER_LEAF){
398         // if leaf node, stop recursing and print something.
399         for (uint i = 0; i < indent; i++) {
400             printf("  ");
401         }
402         printf("Leaf %u : %u triangle(s)", collider->id,
collider->data.leaf.num_vertices / 3);
403         printf(" %f %f %f\n", EXPAND_VEC3(collider-
>obb.half_extents));
404     }
405 }
406
407 /**
408 * Create a collider structure given a body. The body must be
initialised and contain the vertex and index data for the mesh.
409 * @param[in] body The body to calculate the collider for.
410 * @param[out] collider A pointer to where the collider
structure pointer should be written to.
411 * @throws MEMORY_EXCEPTION if malloc() fails.
412 * @throws FAILURE if there was an exception during
calculations.
413 */
414 exception tekCreateCollider(const TekBody* body, TekCollider*
collider) {
415     // algorithm process:
416     // initialise a stack of OBBs
417     // determine the best axis to most equally divide the
vertices of the object
418     // determine the smallest OBB that can fit all the vertices
419     // do for each side and add the OBB to the stack
420     // repeat, dequeue the OBB stack, divide the vertices
inside the OBB in half, recreate
421     // if OBB is not divisible, add a leaf node (triangle)

```

```

422
423     // just set up some vectors and whatever
424     Vector collider_stack = {};
425     tekChainThrow(vectorCreate(16, sizeof(TekColliderNode*),
426     &collider_stack));
426
427     Vector triangles = {};
428     tekChainThrowThen(tekGenerateTriangleArray(body->vertices,
429     body->num_vertices, body->indices, body->num_indices, &triangles), {
430         vectorDelete(&collider_stack);
431     });
432
433     // buffer containing indices of triangles
434     TekColliderNode* collider_node;
435     uint node_id = 0;
436     uint* all_indices = (uint*)malloc(triangles.length *
437     sizeof(uint));
438     if (!all_indices) {
439         vectorDelete(&triangles);
440         vectorDelete(&collider_stack);
441         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
442         for indices.");
443     }
444
445     // bogus code, basically just tell the create collider node
446     // function to use every index in the triangle vector
447     for (uint i = 0; i < triangles.length; i++) {
448         all_indices[i] = i;
449     }
450
451     tekChainThrowThen(tekCreateColliderNode(COLLIDER_NODE,
452     node_id++, &triangles, all_indices, triangles.length,
453     &collider_node), {
454         vectorDelete(&triangles);
455         vectorDelete(&collider_stack);
456     });
457
458     // add initial node to the stack so we can start traversing
459     *collider = collider_node;
460     tekChainThrowThen(vectorAddItem(&collider_stack,
461     &collider_node), {
462         tekColliderCleanup();
463     });

```

```

457
458     while (vectorPopItem(&collider_stack, &collider_node)) {
459         // create an indices buffer, this will organise indices
460         // into left and right of the dividing axis.
461         uint* indices_buffer = (uint*)malloc(collider_node-
462 >num_indices * sizeof(uint));
463         if (!indices_buffer) {
464             tekColliderCleanup();
465             tekThrow(MEMORY_EXCEPTION, "Failed to allocate
466 buffer for indices.");
467         }
468
469
470         // starting with an impossible axis (3), so if it
471         // doesn't change we can tell that the polygons are indivisible.
471         uint axis = 3;
472         uint num_left = 0, num_right = 0;
473
474         // check each axis and count the number of triangles
475         // left and right of the dividing axis.
476         for (uint i = 0; i < 3; i++) {
477             num_left = 0;
478             num_right = 0;
479             const uint num_indices = collider_node-
480 >num_indices;
481             for (uint j = 0; j < num_indices; j++) {
482                 const uint index = collider_node->indices[j];
483                 const struct Triangle* triangle;
484                 vectorGetItemPtr(&triangles, index, &triangle);
485                 vec3 delta;
486                 glm_vec3_sub(triangle->centroid, collider_node-
487 >obb.centre, delta);
488                 const float side = glm_vec3_dot(delta,
489 collider_node->obb.axes[i]);
490                 if (side < 0) {
491                     indices_buffer[num_left] = index;
492                     num_left++;
493                 } else {
494                     indices_buffer[num_indices - num_right - 1]
495 = index;
496                     num_right++;
497                 }
498             }

```

```

490             // once a divisible axis is found (e.g. not all on
the one side) then stop.
491             if ((num_left != 0) && (num_right != 0)) {
492                 axis = i;
493                 break;
494             }
495         }
496         if (axis == 3) {
497             // indivisible - overwrite current node to be a
leaf
498             collider_node->type = COLLIDER_LEAF;
499             const uint num_indices = collider_node-
>num_indices;
500             const uint num_vertices = num_indices * 3;
501             vec3* vertices = (vec3*)malloc(num_vertices *
sizeof(vec3));
502             if (!vertices) {
503                 tekColliderCleanup();
504                 tekThrow(MEMORY_EXCEPTION, "Failed to allocate
memory for vertices.");
505             }
506             vec3* w_vertices = (vec3*)malloc(num_vertices *
sizeof(vec3));
507             if (!w_vertices) {
508                 tekColliderCleanup();
509                 tekThrow(MEMORY_EXCEPTION, "Failed to allocate
memory for world space vertices.");
510             }
511             for (uint i = 0; i < num_indices; i++) {
512                 const struct Triangle* triangle;
513                 const uint index = collider_node->indices[i];
514                 vectorGetItemPtr(&triangles, index, &triangle);
515                 glm_vec3_copy(triangle->vertices[0], vertices[i
* 3]);
516                 glm_vec3_copy(triangle->vertices[1], vertices[i
* 3 + 1]);
517                 glm_vec3_copy(triangle->vertices[2], vertices[i
* 3 + 2]);
518                 for (uint j = 0; j < 3; j++) {
519                     glm_vec3_copy(vertices[i * 3 + j],
w_vertices[i * 3 + j]);
520                 }
521             }

```

```

522
523         collider_node->data.leaf.num_vertices =
num_vertices;
524         collider_node->data.leaf.vertices = vertices;
525         collider_node->data.leaf.w_vertices = w_vertices;
526     } else {
527         // divisible, create two new nodes and set as
children.
528         for (uint right = 0; right < 2; right++) {
529             uint* indices = right ?
&indices_buffer[num_left] : indices_buffer;
530             const uint num_indices = right ? num_right :
num_left;
531             TekColliderNode* new_collider_node;
532
tekChainThrowThen(tekCreateColliderNode(COLLIDER_NODE, node_id++,
&triangles, indices, num_indices, &new_collider_node), {
533                 tekColliderCleanup();
534             });
535
536             if (right) {
537                 collider_node->data.node.right =
new_collider_node;
538             } else {
539                 collider_node->data.node.left =
new_collider_node;
540             }
541
tekChainThrowThen(vectorAddItem(&collider_stack,
&new_collider_node), {
542                 tekColliderCleanup();
543             });
544         }
545     }
546 }
547
548     return SUCCESS;
549 }
550
551 /**
552 * Delete a collider node
553 * @param collider_node The pointer to be freed.

```

```

554     * @param handedness Whether the node was a left or right child
555     * (use LEFT or RIGHT, assume LEFT for root node).
556     static void tekDeleteColliderNode(TekColliderNode*
557     collider_node, const flag handedness) {
558         if (collider_node->type == COLLIDER_LEAF)
559             free(collider_node->data.leaf.vertices);
560         // the right hand node has half the indices buffer, but
561         // left and right indices are allocated as one block. Left has index 0
562         // as the start. Right has index 0 + num_left as the start
563         if (handedness == LEFT)
564             free(collider_node->indices);
565         free(collider_node);
566     }
567
568     struct TekNodeHandednessPair {
569         TekColliderNode* collider_node;
570         flag handedness;
571     };
572
573 /**
574     * Delete a collider node by freeing all allocated memory. Will
575     * set pointer to NULL to avoid misuse of freed pointer.
576     * @param collider The collider structure to free.
577     */
578     void tekDeleteCollider(TekCollider* collider) {
579         if (!collider || !(*collider)) return;
580         Vector collider_stack = {};
581         if (vectorCreate(16, sizeof(struct TekNodeHandednessPair),
582 &collider_stack) != SUCCESS) return;
583
584         // traverse tree and record the handedness of each child
585         // node for when freeing.
586         struct TekNodeHandednessPair pair = {*collider, LEFT};
587         vectorAddItem(&collider_stack, &pair);
588         while (vectorPopItem(&collider_stack, &pair)) {
589             // post-order traversal so that child nodes can be
590             // recorded before freeing the parent.
591             if (pair.collider_node->type == COLLIDER_NODE) {
592                 // temp struct so that the data can be copied into
593                 // the vector
594                 struct TekNodeHandednessPair temp_pair = {};
595
596                 if (collider->type == COLLIDER_NODE) {
597                     // copy data from parent to child
598                     ...
599                 }
600
601                 if (collider->type == COLLIDER_LEAF) {
602                     // copy data from parent to child
603                     ...
604                 }
605
606                 temp_pair.collider_node = pair.collider_node;
607                 temp_pair.handedness = pair.handedness;
608                 vectorAddItem(&collider_stack, &temp_pair);
609             }
610         }
611     }

```

```

587         temp_pair.collider_node = pair.collider_node-
>data.node.right;
588         temp_pair.handedness = RIGHT;
589         vectorAddItem(&collider_stack, &temp_pair);
590         temp_pair.collider_node = pair.collider_node-
>data.node.left;
591         temp_pair.handedness = LEFT;
592         vectorAddItem(&collider_stack, &temp_pair);
593     }
594     tekDeleteColliderNode(pair.collider_node,
pair.handedness);
595 }
596
597 // free vector used to iterate.
598 vectorDelete(&collider_stack);
599 *collider = 0;
600 }
601
602 /**
603 * Update an OBB based on a transformation matrix, so the OBB's
coordinates correspond to world coordinates not local coordinates.
604 * @param obb The OBB to transform
605 * @param transform The matrix to transform with.
606 */
607 void tekUpdateOBB(struct OBB* obb, mat4 transform) {
608     // transform the centre
609     glm_mat4_mulv3(transform, obb->centre, 1.0f, obb-
>w_centre);
610
611     // now transform the axes and half extents
612     for (uint i = 0; i < 3; i++) {
613         // using w=0 to represent vector transformation
614         // scale factor of half extents = scale factor of axes
615         glm_mat4_mulv3(transform, obb->axes[i], 0.0f, obb-
>w_axes[i]);
616         const float scale_factor = glm_vec3_norm(obb->axes[i]);
617         glm_vec3_scale(obb->axes[i], 1.0f / scale_factor, obb-
>axes[i]);
618         obb->w_half_extents[i] = obb->half_extents[i] *
scale_factor;
619     }
620 }
621

```

```

622 /**
623  * Update a leaf node of the collider structure by transforming
624  * each vertex by the transform of the object it relates to.
625  * @param leaf The leaf to transform.
626  * @param transform The transform matrix to use.
627 */
628 void tekUpdateLeaf(TekColliderNode* leaf, mat4 transform) {
629     for (uint i = 0; i < leaf->data.leaf.num_vertices; i++) {
630         // multiply vec3 by mat4, using w=1.0 to represent a
631         // position
632         glm_mat4_mulv3(transform, leaf->data.leaf.vertices[i],
633                         leaf->data.leaf.w_vertices[i]);
634     }
635 }
```

tekphys/collider.h

```

1 #pragma once
2
3 #include "../tekgl.h"
4 #include "../core/exception.h"
5 #include "../core/vector.h"
6
7 #include <cglm/vec3.h>
8
9 #define COLLIDER_LEAF 0
10 #define COLLIDER_NODE 1
11
12 struct TekBody;
13 typedef struct TekBody TekBody;
14
15 struct Triangle {
16     vec3 vertices[3];
17     vec3 centroid;
18     float area;
19 };
20
21 struct OBB {
22     vec3 centre;
23     vec3 axes[3];
24     float half_extents[3];
25
26     vec3 w_centre;
27     vec3 w_axes[3];
```

```

28     float w_half_extents[3];
29 }
30
31 typedef struct TekColliderNode {
32     flag type;
33     uint id;
34     struct OBB obb;
35     uint* indices;
36     uint num_indices;
37     union {
38         struct {
39             struct TekColliderNode* left;
40             struct TekColliderNode* right;
41         } node;
42         struct {
43             vec3* vertices;
44             vec3* w_vertices;
45             uint num_vertices;
46         } leaf;
47     } data;
48 } TekColliderNode;
49
50 typedef TekColliderNode* TekCollider;
51
52 exception tekCreateCollider(const TekBody* body, TekCollider* collider);
53 void tekDeleteCollider(TekCollider* collider);
54
55 void tekUpdateOBB(struct OBB* obb, mat4 transform);
56 void tekUpdateLeaf(TekColliderNode* leaf, mat4 transform);

```

tekphys/collisions.c

```

1 #include "collisions.h"
2
3 #include <stdio.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <cglm/cam.h>
7 #include <cglm/mat4.h>
8
9 #include "body.h"
10 #include "collider.h"
11 #include "geometry.h"

```

```

12 #include "../core/vector.h"
13 #include "../tekgl/manager.h"
14 #include "../core/bitset.h"
15 #include "../stb/stb_image.h"
16
17 #define NOT_INITIALISED 0
18 #define INITIALISED 1
19 #define DE_INITIALISED 2
20
21 #define EPSILON 1e-6f
22 #define EPSILON_SQUARED 1e-12f
23 #define LEFT 0
24 #define RIGHT 1
25
26 struct TekPolytopeVertex {
27     vec3 a;
28     vec3 b;
29     vec3 support;
30 };
31
32 static Vector collider_buffer = {};
33 static Vector contact_buffer = {};
34 static Vector impulse_buffer = {};
35
36 static Vector vertex_buffer = {};
37 static Vector face_buffer = {};
38 static Vector edge_buffer = {};
39 static BitSet edge_bitset = {};
40
41 static flag collider_init = NOT_INITIALISED;
42
43 /**
44 * Called at the end of the program to free any allocated
45 structures.
46 */
47 void tekColliderDelete() {
48     // mark de initialised so that they are not used.
49     collider_init = DE_INITIALISED;
50     // stuff for broad collision detection
51     vectorDelete(&collider_buffer);
52     vectorDelete(&contact_buffer);
53     // collision response
54     vectorDelete(&impulse_buffer);

```

```

54     // GJK + EPA
55     vectorDelete(&vertex_buffer);
56     vectorDelete(&face_buffer);
57     vectorDelete(&edge_buffer);
58     bitsetDelete(&edge_bitset);
59 }
60
61 /**
62  * Initialise some supporting data structures that are needed
63  * by many of the methods used in the collision detection process.
64 */
65 tek_init TekColliderInit() {
66     // needed for collision detection
67     exception tek_exception = vectorCreate(16, 2 *
68     sizeof(TekColliderNode*), &collider_buffer);
69     if (tek_exception != SUCCESS) return;
70
71     tek_exception = vectorCreate(8,
72     sizeof(TekCollisionManifold), &contact_buffer);
73     if (tek_exception != SUCCESS) return;
74
75     // collision response
76     tek_exception = vectorCreate(1, NUM_CONSTRAINTS *
77     sizeof(float), &impulse_buffer);
78     if (tek_exception != SUCCESS) return;
79
80     tek_exception = vectorCreate(4, sizeof(struct
81     TekPolytopeVertex), &vertex_buffer);
82     if (tek_exception != SUCCESS) return;
83
84     tek_exception = vectorCreate(3, 2 * sizeof(uint),
85     &face_buffer);
86     if (tek_exception != SUCCESS) return;
87
88     tek_exception = bitsetCreate(6 * 6, 1, &edge_bitset);
89     if (tek_exception != SUCCESS) return;
90
91     // end of program callback

```

```

90     tek_exception = tekAddDeleteFunc(tekColliderDelete);
91     if (tek_exception != SUCCESS) return;
92
93     collider_init = INITIALISED;
94 }
95
96 /**
97  * Check whether there is a collision between two OBBs using
the separating axis theorem.
98  * @param obb_a The first obb.
99  * @param obb_b The obb that will be tested against the first
one.
100  * @return 1 if there was a collision, 0 otherwise.
101 */
102 static flag tekCheckOBBCollision(struct OBB* obb_a, struct
OBB* obb_b) {
103     // OBB-OBB collision using the separating axis theorem
(SAT)
104     // based on Gottschalk et al., "OBBTree" (1996)
https://www.cs.unc.edu/techreports/96-013.pdf
105     // also described in Ericson, "Real-Time Collision
Detection", Ch. 4
106
107     // algorithm based on
https://jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20oriented%20Bounding%20Boxes.pdf
108     // see section "Optimized Computation of OBBs
Intersections"
109
110     // find the vector between the centres
111
112
113     vec3 translate;
114     glm_vec3_sub(obb_b->w_centre, obb_a->w_centre, translate);
115
116     // some buffers to store some precalculated data that is
reused throughout.
117     float t_array[3];
118     float dot_matrix[3][3];
119
120     for (uint i = 0; i < 3; i++) {
121         t_array[i] = glm_vec3_dot(translate, obb_a-
>w_axes[i]);

```

```

122         for (uint j = 0; j < 3; j++) {
123             dot_matrix[i][j] = glm_vec3_dot(obb_a->w_axes[i],
124                                             obb_b->w_axes[j]) + EPSILON;
125         }
126     }
127     // simple cases - check axes aligned with faces of the
128     // first obb.
129     for (uint i = 0; i < 3; i++) {
130         float mag_sum = 0.0f;
131         for (uint j = 0; j < 3; j++) {
132             mag_sum += fabsf(obb_b->w_half_extents[j] *
133                             dot_matrix[i][j]);
134         }
135     }
136 }
137
138     // medium cases - check axes aligned with the faces of the
139     // second obb.
140     for (uint i = 0; i < 3; i++) {
141         float mag_sum = 0.0f;
142         for (uint j = 0; j < 3; j++) {
143             mag_sum += fabsf(obb_a->w_half_extents[j] *
144                             dot_matrix[j][i]);
145         }
146         const float projection = fabsf(glm_vec3_dot(translate,
147                                                       obb_b->w_axes[i]));
148         if (projection > obb_b->w_half_extents[i] + mag_sum) {
149             return 0;
150         }
151     }
152     // store indices of t_array to access depending on
153     // iteration count
154     const uint multis_indices[3][2] = {
155         {2, 1},
156         {0, 2},
157         {1, 0}
158     };
159

```

```

157      // store indices for direction (W, H or D) for inner and
outer loop cycles
158      const uint cyc_indices[3][2] = {
159          {1, 2},
160          {0, 2},
161          {0, 1}
162      };
163
164      // tricky tricky cases - check axes aligned with the edges
and corners of both - cross products of the axes from before.
165      for (uint i = 0; i < 3; i++) {
166          for (uint j = 0; j < 3; j++) {
167              const uint ti_a = multis_indices[i][0], ti_b =
multis_indices[i][1];
168              const float cmp_base = t_array[ti_a] *
glm_vec3_dot(obb_a->w_axes[ti_b], obb_b->w_axes[j]);
169              const float cmp_subt = t_array[ti_b] *
glm_vec3_dot(obb_a->w_axes[ti_a], obb_b->w_axes[j]);
170              const float cmp = fabsf(cmp_base - cmp_subt);
171
172              const uint cyc_ll = cyc_indices[i][0], cyc_lh =
cyc_indices[i][1];
173              const uint cyc_sl = cyc_indices[j][0], cyc_sh =
cyc_indices[j][1];
174              float tst = 0.0f;
175              tst += fabsf(obb_a->w_half_extents[cyc_ll] *
dot_matrix[cyc_lh][j]);
176              tst += fabsf(obb_a->w_half_extents[cyc_ll] *
dot_matrix[cyc_ll][j]);
177              tst += fabsf(obb_b->w_half_extents[cyc_sl] *
dot_matrix[i][cyc_sh]);
178              tst += fabsf(obb_b->w_half_extents[cyc_sh] *
dot_matrix[i][cyc_sl]);
179
180              if (cmp > tst) {
181                  return 0;
182              }
183          }
184      }
185
186      // if no separation is found in any axis, there must be a
collision.
187      return 1;

```

```

188 }
189
190 /**
191 * Create a transformation matrix that will convert an OBB
192 into an AABB centred around the origin.
193 * @param obb The OBB to create the matrix for.
194 * @param transform The outputted transformation matrix.
195 */
196 static void tekCreateOBBTransform(const struct OBB* obb, mat4
transform) {
197     // some matrix magic.
198     // we know the AABB we want, and the OBB we have, and its
199     // easy to say how to change an AABB to get here.
200     mat4 temp_transform = {
201         obb->w_axes[0][0], obb->w_axes[0][1], obb->w_axes[0]
202 [2], 0.0f,
203         obb->w_axes[1][0], obb->w_axes[1][1], obb->w_axes[1]
204 [2], 0.0f,
205         obb->w_axes[2][0], obb->w_axes[2][1], obb->w_axes[2]
206 [2], 0.0f,
207         obb->w_centre[0], obb->w_centre[1], obb->w_centre[2],
208 1.0f
209     };
210     // invert the matrix to go back the other way.
211     glm_mat4_inv_fast(temp_transform, transform);
212 }
213 /**
214 * Check for a collision between an AABB and a triangle. AABB
215 is assumed to be centred around the origin, with extends of +/- each
216 half extent in that axis.
217 * @param half_extents[3] The half extents of the AABB.
218 * @param triangle[3] The triangle to test against.
219 * @return 1 if there was a collision, 0 otherwise.
220 */
221 static flag tekCheckAABBTriangleCollision(const float
half_extents[3], vec3 triangle[3]) {
222     vec3 face_vectors[3];
223     for (uint i = 0; i < 3; i++) {
224         glm_vec3_sub(
225             triangle[(i + 1) % 3],
226             triangle[i],
227             face_vectors[i]

```

```

221         );
222     }
223
224     vec3 xyz_axes[3] = {
225         {1.0f, 0.0f, 0.0f},
226         {0.0f, 1.0f, 0.0f},
227         {0.0f, 0.0f, 1.0f}
228     };
229
230     // first, test the axis which is the normal of the
triangle.
231     // dots[0] = the distance of the triangle from the centre
of the aabb when projected onto the normal.
232     // face_projection = half the length of the aabb when
projected onto the normal axis
233     //
234     // +---+      /|
235     // |   |      / |
236     // +---+      /__|
237     //
238     // |==|=|-----|-      < this is the axis being projected
against
239     //     ^     ^
240     //     a     b     c
241     // a = face_projection - the aabb projected onto the axis
242     // b = the separation (Separating Axis Theorem)
243     // c = dots[0] - the triangle which is perpendicular to
the axis, projection is a single point.
244     // hence, when c > a, there is separation and no
collision. if a > c, then there is no separation in this axis,
continue the search
245     vec3 face_normal;
246     // normal of a triangle = cross product of any two edges
247     glm_vec3_cross(face_vectors[0], face_vectors[1],
face_normal);
248     glm_vec3_normalize(face_normal);
249     // when testing the face normal axis, all three points of
the triangle project to the same point. so just check point 0
250
251     const float tst = fabsf(glm_vec3_dot(triangle[0],
face_normal));
252     float cmp = 0.0f;
253     for (uint i = 0; i < 3; i++) {

```

```

254         cmp += half_extents[i] * fabsf(face_normal[i]);
255     }
256     if (tst > cmp)
257         return 0;
258
259     // testing against the 3 axes of the AABB, which are by
definition the X, Y and Z axes.
260     float dots[3];
261     for (uint i = 0; i < 3; i++) {
262         for (uint j = 0; j < 3; j++) {
263             dots[j] = glm_vec3_dot(triangle[j], xyz_axes[i]);
264         }
265         // the projection of the AABB along this axis will
just be the half extent:
266         //
267         // +-----+
268         // |      0--->|  this vector represents an axis, as
you can see the axis aligns with the half extent.
269         // +-----+
270
271         const float tri_min = fminf(dots[0], fminf(dots[1],
dots[2]));
272         const float tri_max = fmaxf(dots[0], fmaxf(dots[1],
dots[2]));
273         if (tri_min > half_extents[i] || tri_max < -
half_extents[i])
274             return 0;
275     }
276
277     // testing against the final 9 axes, which are the cross
products against the AABB's axes and the triangle's edge vectors.
278     for (uint i = 0; i < 3; i++) {
279         for (uint j = 0; j < 3; j++) {
280             vec3 curr_axis;
281             glm_vec3_cross(xyz_axes[i], face_vectors[j],
curr_axis);
282             float projection = 0.0f;
283             for (uint k = 0; k < 3; k++) {
284                 projection += half_extents[k] *
fabsf(glm_vec3_dot(xyz_axes[k], curr_axis));
285                 dots[k] = glm_vec3_dot(triangle[k],
curr_axis);
286             }

```

```

287
288         const float tri_min = fminf(dots[0],
fminf(dots[1], dots[2]));
289         const float tri_max = fmaxf(dots[0],
fmaxf(dots[1], dots[2]));
290         if (tri_min > projection || tri_max < -
projection)
291             return 0;
292         }
293     }
294
295     // assuming no separation was found in any axis, then
there must be a collision.
296     return 1;
297 }
298
299 /**
300 * Check for a collision between a triangle and a single OBB.
Works by transforming both into a position where the OBB can be
treated as an AABB (axis aligned bounding box) to reduce
calculation.
301 * @param obb The obb to test against.
302 * @param triangle[3] The triangle to test.
303 * @return 1 if there was a collision, 0 otherwise.
304 */
305 static flag tekCheckOBBTriangleCollision(const struct OBB*
obb, vec3 triangle[3]) {
306     // create the transform that will convert the OBB into an
AABB
307     mat4 transform;
308     tekCreateOBBTTransform(obb, transform);
309
310     vec3 transformed_triangle[3];
311
312     // apply tranform to each point on the triangle.
313     for (uint i = 0; i < 3; i++) {
314         glm_mat4_mulv3(transform, triangle[i], 1.0f,
transformed_triangle[i]);
315     }
316
317     // treat OBB as an AABB.
318     return tekCheckAABBTriangleCollision(obb->w_half_extents,
transformed_triangle);

```

```

319 }
320
321 /**
322 * Check for a collision between an array of triangles and an
323 * OBB.
324 * @param obb The obb to test for collisions.
325 * @param triangles The array of triangles to test.
326 * @param num_triangles The number of triangles (number of
327 * points / 3)
328 * @return 1 if there was a collision between any of the
329 * triangles and the obb, 0 otherwise.
330 */
331 static flag tekCheckOBBTrianglesCollision(const struct OBB*
332 obb, vec3* triangles, const uint num_triangles) {
333     // loop through each triangle
334     for (uint i = 0; i < num_triangles; i++) {
335         // cheeky maths to adjust from triangles to points.
336         if (tekCheckOBBTriangleCollision(obb, triangles + i *
337 )) return 1;
338     }
339     return 0;
340 }
341
342 /**
343 * Calculate the 'orientation' of a triangle versus a point,
344 * it is the scalar triple product of the triangle translated by the
345 * position vector of the point.
346 * @param triangle[3] The triangle to find the orientation of.
347 * @param point The point to find the orientation of.
348 * @return The orientation of the 4 points.
349 */
350 static float tekCalculateOrientation(vec3 triangle[3], vec3
351 point) {
352     // buffer to store difference between each point and the
353     // 4th point.
354     vec3 sub_buffer[3];
355     for (uint i = 0; i < 3; i++) {
356         glm_vec3_sub(triangle[i], point, sub_buffer[i]);
357     }
358     return scalarTripleProduct(sub_buffer[0], sub_buffer[1],
359     sub_buffer[2]);
360 }
361

```

```

352 /**
353  * Return the furthest point on a triangle in a given search
354  * direction.
355  * @param[in] triangle[3] The triangle to search through.
356  * @param[in] direction The direction to search in.
357  * @return closest_point The index of the closest point on the
358  * triangle.
359 */
360 static uint tekTriangleFurthestPoint(vec3 triangle[3], vec3
361 direction) {
362     // track which vertex is the best.
363     uint max_index = 0;
364     float max_value = -FLT_MAX;
365
366     // for each vertex, project onto the direction to find
367     // which is furthest along axis.
368     for (uint i = 0; i < 3; i++) {
369         const float new_max_value = glm_vec3_dot(triangle[i],
370 direction);
371
372         if (new_max_value > max_value) {
373             max_index = i;
374             max_value = new_max_value;
375         }
376     }
377     return max_index;
378 }
379 /**
380  * Triangle support function. Finds a point of a Minkowski
381  * Difference that has the largest magnitude in a specified direction.
382  * @param[in] triangle_a[3] The first triangle to check.
383  * @param[in] triangle_b[3] The second triangle to check.
384  * @param[in] direction The direction to check in
385  * @param[out] point The outputted point of the Minkowski
386  * Difference.
387 */
388 static void tekTriangleSupport(vec3 triangle_a[3], vec3
389 triangle_b[3], vec3 direction, struct TekPolytopeVertex* point) {
390     // find the maximal point of triangle a, and the maximal
391     // point of triangle b in the other direction

```

```

385      // difference of these points will be the largest possible
separation, and give the largest minkowski difference.
386
387      // find the opposite direction
388      vec3 opposite_direction;
389      glm_vec3_negate_to(direction, opposite_direction);
390
391      // find maximum points in this direction
392      const uint index_a = tekTriangleFurthestPoint(triangle_a,
direction);
393      const uint index_b = tekTriangleFurthestPoint(triangle_b,
opposite_direction);
394
395      // subtract to find minkowski difference.
396      glm_vec3_sub(triangle_a[index_a], triangle_b[index_b],
point->support);
397
398      glm_vec3_copy(triangle_a[index_a], point->a);
399      glm_vec3_copy(triangle_b[index_b], point->b);
400  }
401
402 /**
403 * Update a simplex which has two points.
404 * @param[out] direction The direction which is being tested.
405 * @param[in/out] simplex[4] The simplex to update.
406 * @return
407 */
408 static void tekUpdateLineSimplex(vec3 direction, struct
TekPolytopeVertex simplex[4]) {
409     // the new direction will be perpendicular to the line
segment in the direction of the origin.
410     // the simplex cannot be reduced, if point A was closer to
origin it would not pass over the origin, so would be rejected
411     // if B was closer, then A would not have been a possible
point.
412     vec3 ao;
413     glm_vec3_negate_to(simplex[0].support, ao);
414     vec3 ab;
415     glm_vec3_sub(simplex[1].support, simplex[0].support, ab);
416     glm_vec3_cross(ab, ao, direction);
417     glm_vec3_cross(direction, ab, direction);
418 }
419

```

```

420  /**
421   * Helper function to remove duplicate code for case involving
422   * a line segment AB
423   * @param ao[in] The vector from simplex[0] to origin.
424   * @param ab[in] The vector from simplex[0] to simplex[1]
425   * @param direction[out] The new direction to check next time.
426   * @param len_simplex[out] A pointer to the length of the
427   * simplex.
428   */
429 static void tekUpdateTriangleSimplexLineAB(vec3 ao, vec3 ab,
430 vec3 direction, uint* len_simplex) {
431     // is the line AB in the direction of the origin?
432     if (glm_vec3_dot(ab, ao) > 0.0f) {
433         // new direction is perpendicular to AB towards origin
434         glm_vec3_cross(ab, ao, direction);
435         glm_vec3_cross(direction, ab, direction);
436     } else {
437         // new direction is pointing from A to origin.
438         glm_vec3_copy(ao, direction);
439
440         // update simplex, removing point C
441         *len_simplex = 2;
442     }
443 }
444
445 /**
446  * Update a simplex which has three points. The final simplex
447  * could have between 1 and 3 points, and direction will be changed
448  * based on features of simplex.
449  * @param[out] direction The new direction of the simplex.
450  * @param[in/out] simplex[4] The simplex to update.
451  * @param[in/out] len_simplex The length of the simplex.
452  * @return
453  */
454 static void tekUpdateTriangleSimplex(vec3 direction, struct
455 TekPolytopeVertex simplex[4], uint* len_simplex) {
456     // common vectors, used throughout
457     vec3 ao;
458     glm_vec3_negate_to(simplex[0].support, ao);
459     vec3 ab;

```

```

457     glm_vec3_sub(simplex[1].support, simplex[0].support, ab);
458     vec3 ac;
459     glm_vec3_sub(simplex[2].support, simplex[0].support, ac);
460
461     // find normal of triangle
462     vec3 triangle_normal;
463     glm_vec3_cross(ab, ac, triangle_normal);
464
465     // vector perpendicular to line AC and triangle normal
466     vec3 perp_ac;
467     glm_vec3_cross(triangle_normal, ac, perp_ac);
468
469     // is the perpendicular line to AC facing the origin?
470     if (glm_vec3_dot(perp_ac, ao) > 0.0f) {
471         // is the line AC itself facing the origin?
472         if (glm_vec3_dot(ac, ao) > 0.0f) {
473             // new direction is perpendicular to AC towards
origin
474             glm_vec3_cross(ac, ao, direction);
475             glm_vec3_cross(direction, ac, direction);
476
477             // update simplex, removing point B
478             memcpy(&simplex[1], &simplex[2], sizeof(struct
TekPolytopeVertex));
479             *len_simplex = 2;
480         } else {
481             // avoid repeated code by making a small function
482             tekUpdateTriangleSimplexLineAB(ao, ab, direction,
len_simplex);
483         }
484     } else {
485         vec3 perp_ab;
486         glm_vec3_cross(ab, triangle_normal, perp_ab);
487         if (glm_vec3_dot(perp_ab, ao) > 0.0f) {
488             // avoid repeated code by making a small function
489             tekUpdateTriangleSimplexLineAB(ao, ab, direction,
len_simplex);
490         } else {
491             // is the origin above the triangle?
492             if (glm_vec3_dot(triangle_normal, ao) > 0.0f) {
493                 // new direction is above the triangle
494                 glm_vec3_copy(triangle_normal, direction);
495             } else {

```

```

496                         // new direction is below the triangle.
497                         glm_vec3_negate_to(triangle_normal,
direction);
498
499                         // swap winding order so that next test is
consistent
500                         struct TekPolytopeVertex temp;
501                         memcpy(&temp, &simplex[1], sizeof(struct
TekPolytopeVertex));
502                         memcpy(&simplex[1], &simplex[2], sizeof(struct
TekPolytopeVertex));
503                         memcpy(&simplex[2], &temp, sizeof(struct
TekPolytopeVertex));
504                     }
505                 }
506             }
507         }
508
509     /**
510      * Update a simplex with 4 vertices. Tests each valid face as
a separate triangle, and determines if origin is contained.
511      * @param[out] direction The new direction to test
512      * @param[in/out] simplex[4] The simplex to update.
513      * @param[in/out] len_simplex The length of the simplex.
514      * @return 1 if the origin is contained, 0 if not
515     */
516     static int tekUpdateTetrahedronSimplex(vec3 direction, struct
TekPolytopeVertex simplex[4], uint* len_simplex) {
517         // create some commonly used vectors
518         vec3 ao;
519         glm_vec3_negate_to(simplex[0].support, ao);
520         vec3 ab;
521         glm_vec3_sub(simplex[1].support, simplex[0].support, ab);
522         vec3 ac;
523         glm_vec3_sub(simplex[2].support, simplex[0].support, ac);
524
525         // check the triangle abc
526         vec3 norm_abc;
527         glm_vec3_cross(ab, ac, norm_abc);
528
529         // is the origin above this triangle?
530         if (glm_vec3_dot(norm_abc, ao) > 0.0f) {
531             *len_simplex = 3;

```

```

532         tekUpdateTriangleSimplex(direction, simplex,
len_simplex);
533         return 0;
534     }
535
536     // check the triangle acd
537     vec3 ad;
538     glm_vec3_sub(simplex[3].support, simplex[0].support, ad);
539     vec3 norm_acd;
540     glm_vec3_cross(ac, ad, norm_acd);
541
542     // is the origin above this triangle?
543     if (glm_vec3_dot(norm_acd, ao) > 0.0f) {
544         // update vertices so that the first 3 are the
triangle
545         memcpy(&simplex[1], &simplex[2], sizeof(struct
TekPolytopeVertex));
546         memcpy(&simplex[2], &simplex[3], sizeof(struct
TekPolytopeVertex));
547         *len_simplex = 3;
548         tekUpdateTriangleSimplex(direction, simplex,
len_simplex);
549         return 0;
550     }
551
552     // check the triangle adb
553     vec3 norm_adb;
554     glm_vec3_cross(ad, ab, norm_adb);
555
556     // is the origin above this triangle?
557     if (glm_vec3_dot(norm_adb, ao) > 0.0f) {
558         // update vertices
559         memcpy(&simplex[2], &simplex[1], sizeof(struct
TekPolytopeVertex));
560         memcpy(&simplex[1], &simplex[3], sizeof(struct
TekPolytopeVertex));
561         *len_simplex = 3;
562         tekUpdateTriangleSimplex(direction, simplex,
len_simplex);
563         return 0;
564     }
565

```

```

566      // if the origin is not above any of the triangles,
naturally it is below all of them.
567      // this means the origin must be contained within the
tetrahedron, as previous tests have confirmed the origin is also
behind the triangle bcd
568      // hence return true
569
570      return 1;
571 }
572
573 /**
574  * Call the correct update function based on the number of
vertices in the simplex.
575  * @param[out] direction The new direction to travel in.
576  * @param[in/out] simplex[4] The simplex to update.
577  * @param[in/out] len_simplex The number of vertices in the
simplex.
578  * @return 1 if the origin is contained within the simplex, 0
if not or if the simplex is not a tetrahedron.
579 */
580 static int tekUpdateSimplex(vec3 direction, struct
TekPolytopeVertex simplex[4], uint* len_simplex) {
581     switch (*len_simplex) {
582     case 2:
583         // length doesn't change, so ignore the length
584         tekUpdateLineSimplex(direction, simplex);
585         break;
586     case 3:
587         tekUpdateTriangleSimplex(direction, simplex,
len_simplex);
588         break;
589     case 4:
590         // a tetrahedron is required before we can determine
if the simplex contains the origin.
591         return tekUpdateTetrahedronSimplex(direction, simplex,
len_simplex);
592     default:
593         break;
594     }
595     return 0;
596 }
597
598 /**

```

```

599  * Check for a collision between two triangles using GJK
algorithm
600  * @param[in] triangle_a[3] One of the triangles to test.
601  * @param[in] triangle_b[3] The other triangle to be tested
against.
602  * @param[out] simplex[4] The simplex created during the
processing of the triangles. (required to exist beforehand)
603  * @param[out] len_simplex The number of vertices in the
simplex. (required to exist beforehand)
604  * @param[out] separation The seperation of the contact
points.
605  * @return Whether or not the triangles are colliding (0 if
not, 1 if they are)
606  */
607 int tekCheckTriangleCollision(vec3 triangle_a[3], vec3
triangle_b[3], struct TekPolytopeVertex simplex[4], uint*
len_simplex, float* separation) {
608     // to start GJK algorithm, pick an initial direction
609     // start with finding centroid of each triangle
610     vec3 sum_a, sum_b;
611     sumVec3(sum_a, triangle_a[0], triangle_a[1],
triangle_a[2]);
612     sumVec3(sum_b, triangle_b[0], triangle_b[1],
triangle_b[2]);
613
614     // initial direction is vector between the midpoints
615     vec3 direction;
616     glm_vec3_sub(sum_b, sum_a, direction);
617
618     vec3 normal_a, normal_b;
619     triangleNormal(triangle_a, normal_a);
620     triangleNormal(triangle_b, normal_b);
621
622     // if the midpoints overlap then obviously cant use this
vector
623     // or if this vector is parallel to one of the triangles
624     if (fabsf(glm_vec3_norm(direction)) < EPSILON
625         || fabsf(glm_vec3_dot(direction, normal_a)) < EPSILON
626         || fabsf(glm_vec3_dot(direction, normal_b)) < EPSILON)
{
627         vec3 random_direction = {
628             randomFloat(-100.0f, 100.0f), randomFloat(-100.0f,
100.0f), randomFloat(-100.0f, 100.0f)

```

```

629         };
630         glm_vec3_normalize_to(random_direction, direction);
631     }
632
633     // add an initial point to the simplex
634     // find this point by running support function on our
initial direction
635     tekTriangleSupport(triangle_a, triangle_b, direction,
&simplex[0]);
636     *len_simplex = 1;
637
638     // check if the starting point extends past the origin
639     if (glm_vec3_dot(simplex[0].support, direction) < 0.0f)
        return 0;
640
641     // the next direction to check is the opposite to what we
started with
642     // this point is already the maximum of this direction, so
if we kept going this way we'd find nothing
643     glm_vec3_negate(direction);
644
645
646     // find the second point of the simplex.
647     struct TekPolytopeVertex support;
648     tekTriangleSupport(triangle_a, triangle_b, direction,
&support);
649
650     // begin the iterative process
651     // if projecting the new support point against the
direction yields a negative value,
652     // it means that the new support creates a simplex that
does not contain the origin. so no collision.
653     *separation = 0.0f;
654     float sep;
655     glm_vec3_normalize(direction);
656     uint num_iterations = 0;
657     while ((sep = glm_vec3_dot(support.support, direction)) >=
0.0f) {
658         *separation = sep;
659
660         // if point is valid, add it to the start of the
simplex.
661         // shift all the points backwards first

```

```

662         memmove(&simplex[1], &simplex[0], 3 * sizeof(struct
TekPolytopeVertex));
663         memcpy(&simplex[0], &support, sizeof(struct
TekPolytopeVertex));
664         (*len_simplex)++;
665
666         // update simplex. if returns true, then the origin is
contained so we can stop searching
667         if (tekUpdateSimplex(direction, simplex, len_simplex))
668             return 1;
669
670         // if the search direction is nowhere, it means that
we have already passed over the origin
671         if (fabsf(glm_vec3_norm(direction)) < EPSILON) {
672             return 1;
673         }
674
675         // find the next support point.
676         tekTriangleSupport(triangle_a, triangle_b, direction,
&support);
677         glm_vec3_normalize(direction);
678
679         // prevent infinite loop
680         if (num_iterations++ > 20) return 0;
681     }
682
683     return 0;
684 }
685
686 /**
687  * Old test function that survived because I forgot to remove
it. But now its too late.
688  * @param triangle_a[3] The first triangle involved in the
test.
689  * @param triangle_b[3] The second triangle involved in the
test.
690  * @return 1 if there was a collision between them, 0
otherwise. (I presume)
691 */
692 int tekTriangleTest() {
693     vec3 triangle_a[] = {
694         0.46666431427001953f, 1.510340690612793f,
1.214371681213379f,

```

```

695         1.0f, 0.0f, 0.0f,
696         -2.7034502029418945f, -3.8242716789245605f, -
0.3502063751220703f
697     };
698     vec3 triangle_b[] = {
699         -0.5204248428344727f, -0.7979059219360352f, -
0.9570636749267578f,
700         -5.015390872955322f, -4.310789585113525f,
1.0890388488769531f,
701         0.699742317199707f, 0.5617036819458008f,
1.45734441280365f
702     };
703
704     struct TekPolytopeVertex polytope[4] = {};
705     uint len_simplex = 0;
706     float separation = 0.0f;
707
708     return tekCheckTriangleCollision(triangle_a, triangle_b,
&polytope, &len_simplex, &separation);
709 }
710
711 /**
712 * Return the axis of smallest magnitude of a vector.
713 * @param vector The vector to check.
714 * @return 0 if X is the smallest, 1 if Y is the smallest and
2 if Z is the smallest.
715 */
716 static uint tekGetMinAxis(vec3 vector) {
717     // loop through each axis (x, y and z) to find which of
those coordinates has the smallest magnitude
718     uint min_axis = 0;
719     float min_length = FLT_MAX;
720     for (uint i = 0; i < 3; i++) {
721         // if a new smaller axis is found, update.
722         if (fabsf(vector[i]) < min_length) {
723             min_length = fabsf(vector[i]);
724             min_axis = i;
725         }
726     }
727     return min_axis;
728 }
729
730 /**

```

```

731     * Blow up a simplex so that it is 3D - some simplexes can end
when they are a line or a triangle, but the EPA algorithm requires
that there is a tetrahedron.
732     * @param triangle_a[3] The first triangle involved in the
collision detection.
733     * @param triangle_b[3] The second triangle involved in the
collision detection.
734     * @param simplex[4] The current state of the simplex.
735     * @param len_simplex The number of points in the simplex.
736     */
737     static void tekGrowSimplex(vec3 triangle_a[3], vec3
triangle_b[3], struct TekPolytopeVertex simplex[4], uint
len_simplex) {
738         vec3 axes[] = {
739             1.0f, 0.0f, 0.0f, // +X
740             0.0f, 1.0f, 0.0f, // +Y
741             0.0f, 0.0f, 1.0f, // +Z
742             -1.0f, 0.0f, 0.0f, // -X
743             0.0f, -1.0f, 0.0f, // -Y
744             0.0f, 0.0f, -1.0f // -Z
745         };
746
747         vec3 ab, ac, ad, delta, normal, direction;
748         uint min_axis;
749         const float angle = GLM_PI / 3.0f;
750         switch (len_simplex) {
751             case 4:
752                 // find vectors making up the current tetrahedron
753                 glm_vec3_sub(simplex[1].support, simplex[0].support,
ab);
754                 glm_vec3_sub(simplex[2].support, simplex[0].support,
ac);
755                 glm_vec3_sub(simplex[3].support, simplex[0].support,
ad);
756
757                 // get normal of base
758                 glm_vec3_cross(ab, ac, normal);
759
760                 // find volume of the newly created tetrahedron, if
its near zero then reduce to a triangle
761                 glm_vec3_sub(simplex[3].support, simplex[0].support,
ad);
762                 if (fabsf(glm_vec3_dot(ad, normal)) < EPSILON)

```

```

763             len_simplex = 3;
764         else
765             break;
766     case 3:
767         // find two vectors making the edges of triangle
768         glm_vec3_sub(simplex[1].support, simplex[0].support,
769 ab);
770         glm_vec3_sub(simplex[2].support, simplex[0].support,
771 ac);
772         // if collinear, then degenerate, so reduce to a line
773         glm_vec3_cross(ab, ac, normal);
774         if (glm_vec3_norm2(normal) < EPSILON_SQUARED)
775             len_simplex = 2;
776         else
777             break;
778     case 2:
779         // check if line is made of two of the same point
780         glm_vec3_sub(simplex[1].support, simplex[0].support,
781 ab);
782         if (glm_vec3_norm2(ab) < EPSILON_SQUARED)
783             len_simplex = 1;
784         else
785             break;
786     switch (len_simplex) {
787     case 1:
788         // test against each principal direction
789         for (uint i = 0; i < 6; i++) {
790             // find support point in that axis
791             tekTriangleSupport(triangle_a, triangle_b,
792 axes[i], &simplex[1]);
793             // if vector between original point and new point
794             // is not 0 (e.g. not the same point) then use this one.
795             glm_vec3_sub(simplex[1].support,
796 simplex[0].support, ab);
797             if (glm_vec3_norm2(ab) > EPSILON_SQUARED)
798                 break;
799         }
800     case 2:
801         if (len_simplex > 1)

```

```

800             glm_vec3_sub(simplex[1].support,
simplex[0].support, ab);
801             min_axis = tekGetMinAxis(ab);
802             glm_vec3_normalize(ab);
803             glm_vec3_cross(axes[min_axis], ab, direction);
804             for (uint i = 0; i < 6; i++) {
805                 tekTriangleSupport(triangle_a, triangle_b,
direction, &simplex[2]);
806
807                 // if new point is close to the origin
808                 if (glm_vec3_norm2(simplex[2].support) <
EPSILON_SQUARED) {
809                     glm_vec3_rotate(direction, angle, ab);
810                     continue;
811                 }
812
813                 // if point B == point C, reject
814                 glm_vec3_sub(simplex[2].support,
simplex[1].support, delta);
815                 if (glm_vec3_norm2(delta) < EPSILON_SQUARED) {
816                     glm_vec3_rotate(direction, angle, ab);
817                     continue;
818                 }
819
820                 // if point A == point C, reject
821                 glm_vec3_sub(simplex[2].support,
simplex[0].support, delta);
822                 if (glm_vec3_norm2(delta) < EPSILON_SQUARED) {
823                     glm_vec3_rotate(direction, angle, ab);
824                     continue;
825                 }
826
827                 // if ABC are collinear, reject
828                 glm_vec3_cross(ab, delta, normal);
829                 if (glm_vec3_norm2(normal) < EPSILON_SQUARED) {
830                     glm_vec3_rotate(direction, angle, ab);
831                     continue;
832                 }
833                 break;
834             }
835         case 3:
836             // find vectors making up the current triangle

```

```

837         glm_vec3_sub(simplex[1].support, simplex[0].support,
ab);
838         glm_vec3_sub(simplex[2].support, simplex[0].support,
ac);
839
840         // get normal of triangle
841         glm_vec3_cross(ab, ac, direction);
842
843         // search in direction of triangle normal for the next
point
844         tekTriangleSupport(triangle_a, triangle_b, direction,
&simplex[3]);
845
846         // find volume of the newly created tetrahedron, if
its near zero then try opposite direction
847         glm_vec3_sub(simplex[3].support, simplex[0].support,
ad);
848         if (fabsf(glm_vec3_dot(ad, direction)) < EPSILON) {
849             glm_vec3_negate(direction);
850             tekTriangleSupport(triangle_a, triangle_b,
direction, &simplex[3]);
851         }
852     default:
853         break;
854     }
855
856     // ensure that the tetrahedron is wound the right way
857     vec3 da, db, dc;
858     glm_vec3_sub(simplex[0].support, simplex[3].support, da);
859     glm_vec3_sub(simplex[1].support, simplex[3].support, db);
860     glm_vec3_sub(simplex[2].support, simplex[3].support, dc);
861
862     // if the volume of this tetrahedron is negative, it means
that ABC is wound the wrong way
863     glm_vec3_cross(db, dc, normal);
864     if (glm_vec3_dot(da, normal) < 0.0f) {
865         struct TekPolytopeVertex swap_vertex;
866         memcpy(&swap_vertex, &simplex[0], sizeof(struct
TekPolytopeVertex));
867         memcpy(&simplex[0], &simplex[1], sizeof(struct
TekPolytopeVertex));
868         memcpy(&simplex[1], &swap_vertex, sizeof(struct
TekPolytopeVertex));

```

```

869      }
870  }
871
872 /**
873  * Find the closest face of a polytope to the origin.
874  * @param vertices A vector filled with vertices of the
polytope.
875  * @param faces A vector filled with indices of faces.
876  * @param face_index The index of the closest face.
877  * @param face_distance The distance between the face and the
origin.
878  * @param face_normal
879  * @throws VECTOR_EXCEPTION if something goes horribly wrong
e.g. cosmic bit flip
880 */
881 static exception tekGetClosestFace(const Vector* vertices,
const Vector* faces, uint* face_index, float* face_distance, vec3
face_normal) {
882     // default tracker values
883     *face_distance = FLT_MAX;
884     *face_index = 0;
885
886     // iterate over all faces in the polytope
887     for (uint i = 0; i < faces->length; i++) {
888         // get face vertex indices
889         uint* indices;
890         tekChainThrow(vectorGetItemPtr(faces, i, &indices));
891
892         // get the three vertices that make up the face
893         struct TekPolytopeVertex* vertex_a, * vertex_b, *
vertex_c;
894         tekChainThrow(vectorGetItemPtr(vertices, indices[0],
&vertex_a));
895         tekChainThrow(vectorGetItemPtr(vertices, indices[1],
&vertex_b));
896         tekChainThrow(vectorGetItemPtr(vertices, indices[2],
&vertex_c));
897
898         // find the edge vectors
899         vec3 ab;
900         glm_vec3_sub(vertex_b->support, vertex_a->support,
ab);
901         vec3 ac;

```

```

902         glm_vec3_sub(vertex_c->support, vertex_a->support,
ac);
903
904         // cross product = normal
905         vec3 normal;
906         glm_vec3_cross(ab, ac, normal);
907
908         const float mag_normal = glm_vec3_norm(normal);
909         if (mag_normal < EPSILON)
910             continue;
911
912         glm_vec3_divs(normal, mag_normal, normal);
913
914         // we can also compare square distances, because if x
> y, sqrt(x) > sqrt(y)
915         const float curr_distance = fabsf(glm_vec3_dot(normal,
vertex_a->support));
916
917         if (curr_distance < *face_distance) {
918             *face_distance = curr_distance;
919             *face_index = i;
920             glm_vec3_copy(normal, face_normal);
921         }
922     }
923
924     return SUCCESS;
925 }
926
927 /**
928  * Remove an edge from a polytope given the indices of the two
vertices that make up that edge.
929  * @param edges The vector of all edges in the polytope.
930  * @param edges_bitset The bitset containing the presence of
edges in the polytope.
931  * @param indices[2] The indices of the edge to be removed.
932  * @throws VECTOR_EXCEPTION .
933  * @throws BITSET_EXCEPTION .
934 */
935 static exception tekRemoveEdgeFromPolytope(Vector* edges,
BitSet* edges_bitset, const uint indices[2]) {
936     // if the opposite edge already exists in the edge list,
discard this edge and remove

```

```

937      // this is because the two triangles are joined together,
so this edge will disappear from the simplex
938      flag has_opposite_edge;
939      tekChainThrow(bitsetGet2D(edges_bitset, indices[1],
indices[0], &has_opposite_edge));
940      if (has_opposite_edge) {
941          tekChainThrow(bitsetUnset2D(edges_bitset, indices[1],
indices[0]));
942          return SUCCESS;
943      }
944
945      // otherwise, set edge to be extended as new face if not
set already
946      flag has_edge;
947      tekChainThrow(bitsetGet2D(edges_bitset, indices[0],
indices[1], &has_edge));
948      if (!has_edge) {
949          tekChainThrow(bitsetSet2D(edges_bitset, indices[0],
indices[1]));
950          tekChainThrow(vectorAddItem(edges, indices));
951      }
952
953      return SUCCESS;
954 }
955
956 /**
957 * Remove a face from a polytope, and record any edges that
could possibly be removed in the edges vector, and whether or not
they should actually be removed in the edges bitset.
958 * @param faces A vector containing the indices of vertices
for each face in the polytope.
959 * @param edges A vector containing edge vertices
960 * @param edges_bitset A bitset that marks which edges need to
be extended
961 * @param face_index The index of the face to be removed
962 * @throws MEMORY_EXCEPTION if malloc() fails.
963 * @throws VECTOR_EXCEPTION if face_index is out of range.
964 */
965 static exception tekRemoveFaceFromPolytope(Vector* faces,
Vector* edges, BitSet* edges_bitset, const uint face_index) {
966     // remove face and get indices of vertices
967     uint indices[3];

```

```

968     tekChainThrow(vectorRemoveItem(faces, face_index,
969 indices));
970     // now get each pair of indices and remove the associated
edges
971     for (uint i = 0; i < 3; i++) {
972         const uint edge_indices[] = {
973             indices[i], indices[(i + 1) % 3]
974         };
975
976         tekChainThrow(tekRemoveEdgeFromPolytope(edges,
edges_bitset, edge_indices));
977     }
978
979     return SUCCESS;
980 }
981
982 /**
983 * Iterate over faces and remove any which have the support in
their positive halfspace
984 * @param vertices A vector containing all the vertices in the
polytope.
985 * @param faces A vector containing the indices of vertices
for each face.
986 * @param edges A vector containing pairs of indices that
represent edges
987 * @param edges_bitset A bitset that records which edges
should actually be used to create new faces
988 * @param support The support position
989 * @throws MEMORY_EXCEPTION if malloc() fails
990 */
991 static exception tekRemoveAllVisibleFaces(const Vector*
vertices, Vector* faces, Vector* edges, BitSet* edges_bitset, vec3
support) {
992     // iterate over all faces
993     for (uint i = 0; i < faces->length; i++) {
994         uint indices[3];
995         tekChainThrow(vectorGetItem(faces, i, indices));
996
997         // get the vertices of the face
998         vec3 triangle[3];
999         for (uint j = 0; j < 3; j++) {
1000             const uint index = indices[j];

```

```

1001     struct TekPolytopeVertex* vertex;
1002     tekChainThrow(vectorGetItemPtr(vertices, index,
1003     &vertex));
1004     glm_vec3_copy(vertex->support, triangle[j]);
1005
1006     vec3 ab, ac, normal;
1007     glm_vec3_sub(triangle[1], triangle[0], ab);
1008     glm_vec3_sub(triangle[2], triangle[0], ac);
1009     glm_vec3_cross(ab, ac, normal);
1010     const float mag_normal = glm_vec3_norm(normal);
1011     if (mag_normal < EPSILON) {
1012         tekChainThrow(vectorRemoveItem(faces, i, NULL));
1013         i--;
1014         continue;
1015     }
1016     glm_vec3_divs(normal, mag_normal, normal);
1017
1018     vec3 proj;
1019     glm_vec3_sub(support, triangle[0], proj);
1020
1021     // if triangle is "visible" from support point / on
positive halfspace.
1022     const float orientation = glm_vec3_dot(normal, proj);
1023     if (orientation > EPSILON) {
1024         // remove face from list
1025         tekChainThrow(vectorRemoveItem(faces, i, NULL));
1026         i--;
1027
1028         // add edges to construct list.
1029         for (uint j = 0; j < 3; j++) {
1030             const uint edge_indices[] = {
1031                 indices[j], indices[(j + 1) % 3]
1032             };
1033             tekChainThrow(tekRemoveEdgeFromPolytope(edges,
edges_bitset, edge_indices));
1034         }
1035     }
1036 }
1037
1038     return SUCCESS;
1039 }
1040

```

```

1041 /**
1042  * Add a face (3 vertex indices) to a vector filled with
faces. (face_buffer)
1043  * @note Faces are treated with counter-clockwise winding
order. So the "top" of the triangle is the direction where a->b, b-
>c, c->a is an anticlockwise order.
1044  * @param faces The face vector.
1045  * @param index_a The first index.
1046  * @param index_b The second index.
1047  * @param index_c The third index.
1048  * @throws MEMORY_EXCEPTION if malloc() fails.
1049 */
1050 static exception tekAddFaceToPolytope(Vector* faces, const
uint index_a, const uint index_b, const uint index_c) {
1051     // just makes a little array so that i can add it to the
vector as one chunk.
1052     const uint indices[] = {
1053         index_a, index_b, index_c
1054     };
1055     tekChainThrow(vectorAddItem(faces, indices));
1056     return SUCCESS;
1057 }
1058
1059 /**
1060  * Add a face to fill a gap formed by adding a new support
point.
1061  * @param vertices A vector containing the indices of the
polytope.
1062  * @param faces A vector containing the indices of vertices
that make up faces.
1063  * @param edge_indices[2] The indices of the edge vertices
that make a side of the new triangle.
1064  * @param support_index The index of the newly added point.
1065  * @throws MEMORY_EXCEPTION if malloc() fails.
1066 */
1067 static exception tekAddFillerFace(const Vector* vertices,
Vector* faces, const uint edge_indices[2], const uint support_index)
{
1068     // get the three vertices that make it up
1069     struct TekPolytopeVertex* vertex_a, * vertex_b, *
vertex_c;
1070     tekChainThrow(vectorGetItemPtr(vertices, support_index,
&vertex_a));

```

```

1071     tekChainThrow(vectorGetItemPtr(vertices, edge_indices[0],
1072     &vertex_b));
1072     tekChainThrow(vectorGetItemPtr(vertices, edge_indices[1],
1073     &vertex_c));
1073
1074     // find the normal of the new face
1075     vec3 ab, ac;
1076     glm_vec3_sub(vertex_b->support, vertex_a->support, ab);
1077     glm_vec3_sub(vertex_c->support, vertex_a->support, ac);
1078
1079     vec3 normal;
1080     glm_vec3_crossn(ab, ac, normal);
1081
1082     // check if normal is facing the wrong way
1083     if (glm_vec3_dot(normal, vertex_a->support) < 0.0f) {
1084         // wrong way, flip the winding order
1085         tekChainThrow(tekAddFaceToPolytope(faces,
1086 support_index, edge_indices[1], edge_indices[0]));
1087     } else {
1088         // right way, keep the winding order
1089         tekChainThrow(tekAddFaceToPolytope(faces,
1090 support_index, edge_indices[0], edge_indices[1]));
1091     }
1092 }
1093
1094 /**
1095 * If a point is added to a polytope, it produces a "hole" in
1096 the polytope which needs to be filled, which this function does.
1097 * @param vertices The vertices that make up the polytope
1098 * @param faces The faces (triplets of vertex indices) that
1099 * make up the shape.
1100 * @param edges The edges (pairs of indices) that make up the
1101 * shape.
1102 * @param edges_bitset Bitset which marks which indices are
1103 * connected by an edge.
1104 * @param support_index The index of the newly added vertex.
1105 * @throws VECTOR_EXCEPTION .
1106 * @throws BITSET_EXCEPTION .
1107 */

```

```

1104 static exception tekAddAllFillerFaces(const Vector* vertices,
Vector* faces, const Vector* edges, const BitSet* edges_bitset,
const uint support_index) {
1105     // go through all the edges, and add a face for each edge
1106     for (uint i = 0; i < edges->length; i++) {
1107         uint* indices;
1108         tekChainThrow(vectorGetItemPtr(edges, i, &indices));
1109
1110         // not sure why i did that? maybe just wanted an
excuse to use my bitset
1111         flag has_edge;
1112         tekChainThrow(bitsetGet2D(edges_bitset, indices[0],
indices[1], &has_edge));
1113
1114         if (has_edge)
1115             tekChainThrow(tekAddFillerFace(vertices, faces,
indices, support_index));
1116     }
1117
1118     return SUCCESS;
1119 }
1120
1121 /**
1122 * Print out a polytope, this is a collection of vertices in
face groups that form a closed shape / minkowski sum.
1123 * @param vertices
1124 * @param faces
1125 * @throws VECTOR_EXCEPTION .
1126 */
1127 static exception tekPrintPolytope(const Vector* vertices,
const Vector* faces) {
1128     // print out the vertices
1129     for (uint i = 0; i < vertices->length; i++) {
1130         struct TekPolytopeVertex* vertex;
1131         tekChainThrow(vectorGetItemPtr(vertices, i, &vertex));
1132     }
1133
1134     // print out the faces
1135     for (uint i = 0; i < faces->length; i++) {
1136         uint* indices;
1137         tekChainThrow(vectorGetItemPtr(faces, i, &indices));
1138
1139         // get the three vertices that make it up

```

```

1140         struct TekPolytopeVertex* vertex_a, * vertex_b, *
vertex_c;
1141         tekChainThrow(vectorGetItemPtr(vertices, indices[0],
&vertex_a));
1142         tekChainThrow(vectorGetItemPtr(vertices, indices[1],
&vertex_b));
1143         tekChainThrow(vectorGetItemPtr(vertices, indices[2],
&vertex_c));
1144     }
1145
1146     return SUCCESS;
1147 }
1148
1149 /**
1150 * Take a triangle, and project the origin in the direction of
the triangle's face normal. Then get the barycentric coordinates of
where the projection meets the triangle.
1151 * @param point_a The first point on the triangle.
1152 * @param point_b The second point on the triangle.
1153 * @param point_c The third point on the triangle.
1154 * @param barycentric The outputted barycentric coordinates.
1155 */
1156 void tekProjectOriginToBarycentric(vec3 point_a, vec3 point_b,
vec3 point_c, vec3 barycentric) {
1157     // edge vectors
1158     vec3 ab, ac;
1159     glm_vec3_sub(point_b, point_a, ab);
1160     glm_vec3_sub(point_c, point_a, ac);
1161
1162     // face normal = cross product of edge vectors
1163     vec3 normal;
1164     glm_vec3_cross(ab, ac, normal);
1165
1166     // other vectors
1167     vec3 ao;
1168     glm_vec3_negate_to(point_a, ao);
1169
1170     vec3 ab_ao, ao_ac;
1171     glm_vec3_cross(ab, ao, ab_ao);
1172     glm_vec3_cross(ao, ac, ao_ac);
1173
1174     // find square of magnitude
1175     const float square_normal = glm_vec3_dot(normal, normal);

```

```

1176
1177     // formula for barycentric coordinate.
1178     barycentric[2] = glm_vec3_dot(ab_ao, normal) /
square_normal;
1179     barycentric[1] = glm_vec3_dot(ao_ac, normal) /
square_normal;
1180     barycentric[0] = 1 - barycentric[2] - barycentric[1];
1181 }
1182
1183 /**
1184 * Take a triangle plus a barycentric coordinates in terms of
u, v and w, and produce a single point that is specified by the
barycentric coordinate.
1185 * @param point_a The first point on the triangle.
1186 * @param point_b The second point on the triangle.
1187 * @param point_c The third point on the triangle.
1188 * @param barycentric The barycentric coordinate (u,v,w relate
to points a,b,c)
1189 * @param point The outputted point.
1190 */
1191 static void tekCreatePointFromBarycentric(vec3 point_a, vec3
point_b, vec3 point_c, vec3 barycentric, vec3 point) {
1192     // p = u*a + v*b + w*c
1193     glm_vec3_scale(point_a, barycentric[0], point);
1194     glm_vec3_muladds(point_b, barycentric[1], point);
1195     glm_vec3_muladds(point_c, barycentric[2], point);
1196 }
1197
1198 /**
1199 * Find the points of collision between two triangles using
the Expanding Polytope Algorithm (EPA). Uses the
1200 * "waste products" of GJK to begin with, and further
processes this to find the contact normal and points.
1201 * @param triangle_a[3] The first triangle involved in the
collision
1202 * @param triangle_b[3] The second triangle involved in the
collision
1203 * @param simplex[4] The simplex, generated during EPA.
1204 * @param contact_depth Outputted depth of the contact /
displacement between contacts
1205 * @param contact_normal The direction of least penetration
between the triangles.

```

```

1206     * @param contact_a The first point of contact, found on the
1207     first triangle
1208     * @param contact_b The second point of contact, found on the
1209     second triangle.
1210     * @throws FAILURE if supporting data structures were not
1211     initialised.
1212     * @throws VECTOR_EXCEPTION if cosmic bit flip occurs
1213     */
1214     static exception tekGetTriangleCollisionPoints(vec3
1215     triangle_a[3], vec3 triangle_b[3], struct TekPolytopeVertex
1216     simplex[4], float* contact_depth, vec3 contact_normal, vec3
1217     contact_a, vec3 contact_b) {
1218         // general process: generate support point in starting
1219         direction.
1220         // find closest face on the simplex to this point.
1221         // backtrack towards origin and find another support point
1222         // keep going until cannot get closer to the origin - this
1223         is the best contact point
1224
1225         // requires some supporting memory to be allocated.
1226         if (collider_init == NOT_INITIALISED)
1227             tekThrow(FAILURE, "Collider was not initialised.");
1228         if (collider_init == DE_INITIALISED)
1229             return SUCCESS;
1230
1231         // reset all supporting structures for use
1232         vertex_buffer.length = 0;
1233         face_buffer.length = 0;
1234         edge_buffer.length = 0;
1235         bitsetClear(&edge_bitset);
1236
1237         // add vertices from simplex to the vertex buffer.
1238         for (uint i = 0; i < 4; i++) {
1239             tekChainThrow(vectorAddItem(&vertex_buffer,
1240             &simplex[i]));
1241         }
1242
1243         // the four faces of the tetrahedron. we can hard-code the
1244         vertices because GJK guarantees this configuration.
1245         tekChainThrow(tekAddFaceToPolytope(&face_buffer, 0, 1,
1246         2));
1247         tekChainThrow(tekAddFaceToPolytope(&face_buffer, 0, 3,
1248         1));

```

```

1237     tekChainThrow(tekAddFaceToPolytope(&face_buffer, 0, 2,
1238     3));
1239     tekChainThrow(tekAddFaceToPolytope(&face_buffer, 1, 3,
1240     2));
1241     uint face_index;
1242     vec3 face_normal;
1243     float min_distance = FLT_MAX;
1244     uint num_iterations = 0;
1245     while (min_distance == FLT_MAX) {
1246         // clear edge data before next iteration.
1247         edge_buffer.length = 0;
1248         bitsetClear(&edge_bitset);
1249         tekChainThrow(tekGetClosestFace(&vertex_buffer,
1250             &face_buffer, &face_index, &min_distance, face_normal));
1251         // prevent infinite loop
1252         if (num_iterations > 20) {
1253             break;
1254         }
1255         struct TekPolytopeVertex support;
1256         tekTriangleSupport(triangle_a, triangle_b,
1257             face_normal, &support);
1258         if (fabsf(glm_vec3_dot(face_normal, support.support) -
1259             min_distance) < EPSILON)
1260             continue;
1261         min_distance = FLT_MAX;
1262         // remove the closest face
1263         tekChainThrow(tekRemoveFaceFromPolytope(&face_buffer,
1264             &edge_buffer, &edge_bitset, face_index));
1265         // remove any faces that are visible to the support
1266         point
1267         tekChainThrow(tekRemoveAllVisibleFaces(&vertex_buffer,
1268             &face_buffer, &edge_buffer, &edge_bitset, support.support));
1269         const uint support_index = vertex_buffer.length;

```

```

1271         tekChainThrow(vectorAddItem(&vertex_buffer,
1272                                     &support));
1273         tekChainThrow(tekAddAllFillerFaces(&vertex_buffer,
1274                                     &face_buffer, &edge_buffer, &edge_bitset, support_index));
1274         num_iterations++;
1275     }
1276
1277     // at the end, we will have found the closest face to the
1278     // origin, this is the best contact between triangles.
1279     uint* indices;
1280     tekChainThrow(vectorGetItemPtr(&face_buffer, face_index,
1281                                     &indices));
1282
1283     // get the vertices of this face.
1284     struct TekPolytopeVertex* vertex_a, * vertex_b, *
1285     vertex_c;
1286     tekChainThrow(vectorGetItemPtr(&vertex_buffer, indices[0],
1287                                     &vertex_a));
1288     tekChainThrow(vectorGetItemPtr(&vertex_buffer, indices[1],
1289                                     &vertex_b));
1290     tekChainThrow(vectorGetItemPtr(&vertex_buffer, indices[2],
1291                                     &vertex_c));
1292
1293     // now interpolate this face back onto the original
1294     // triangles to get the contact points
1295     vec3 barycentric;
1296     tekProjectOriginToBarycentric(vertex_a->support, vertex_b-
1297                                     >support, vertex_c->support, barycentric);
1298
1299     tekCreatePointFromBarycentric(vertex_a->a, vertex_b->a,
1300                                     vertex_c->a, barycentric, contact_a);
1301     tekCreatePointFromBarycentric(vertex_a->b, vertex_b->b,
1302                                     vertex_c->b, barycentric, contact_b);
1303
1304     glm_vec3_copy(face_normal, contact_normal);
1305     *contact_depth = min_distance;
1306
1307     return SUCCESS;
1308 }
1309
1310 /**

```

```

1301  * Generate a collision manifold between two triangles, if
they are colliding.
1302  * @param triangle_a[3] The three points of the first
triangle.
1303  * @param triangle_b[3] The three points of the second
triangle.
1304  * @param collision Set to 1 if there is a collision, 0
otherwise.
1305  * @param manifold A pointer to a collision manifold that can
have the collision data written into it.
1306  * @throws NOTHING unless I change the function one day.
1307  */
1308 static exception tekGetTriangleCollisionManifold(vec3
triangle_a[3], vec3 triangle_b[3], flag* collision,
TekCollisionManifold* manifold) {
1309     // test for the collision using GJK (Oh yeah)
1310     struct TekPolytopeVertex simplex[4];
1311     uint len_simplex;
1312     float gjk_separation;
1313     if (!tekCheckTriangleCollision(triangle_a, triangle_b,
simplex, &len_simplex, &gjk_separation)) {
1314         *collision = 0;
1315         return SUCCESS;
1316     }
1317     *collision = 1;
1318
1319     // once we're certain there is a collision, blow up the
simplex to use EPA
1320     tekGrowSimplex(triangle_a, triangle_b, simplex,
len_simplex);
1321
1322     vec3 norm_a, norm_b;
1323     triangleNormal(triangle_a, norm_a);
1324     triangleNormal(triangle_b, norm_b);
1325
1326     vec3 cross;
1327     glm_vec3_cross(norm_a, norm_b, cross);
1328
1329     // make sure that we don't have parallel triangles cuz then
it doesn't work
1330     if (glm_vec3_norm2(cross) < EPSILON_SQUARED) {
1331         // ultimate bodge (no mathematical basis)
1332         uint min_a = 0, min_b = 0;

```

```

1333     float min_separation = FLT_MAX;
1334     vec3 separation;
1335     for (uint i = 0; i < 3; i++) {
1336         for (uint j = 0; j < 3; j++) {
1337             vec3 delta;
1338             glm_vec3_sub(triangle_b[j], triangle_a[i],
delta);
1339             const float curr_separation =
glm_vec3_norm2(delta);
1340             if (curr_separation < min_separation) {
1341                 min_separation = curr_separation;
1342                 min_a = i;
1343                 min_b = j;
1344                 glm_vec3_copy(delta, separation);
1345             }
1346         }
1347     }
1348
1349     // doesn't really work well but ^\_(ツ)_/-
1350     manifold->penetration_depth = sqrtf(min_separation);
1351     glm_vec3_scale(separation, -1.0f / manifold-
>penetration_depth, manifold->contact_normal);
1352     glm_vec3_copy(triangle_a[min_a], manifold-
>contact_points[0]);
1353     glm_vec3_copy(triangle_b[min_b], manifold-
>contact_points[1]);
1354 } else {
1355
tekChainThrow(tekGetTriangleCollisionPoints(triangle_a, triangle_b,
simplex,
1356             &manifold->penetration_depth, manifold-
>contact_normal,
1357             manifold->contact_points[0], manifold-
>contact_points[1])
1358         );
1359     }
1360
1361     // somehow this finds a perpendicular vector to the
contact normal?
1362     if (manifold->contact_normal[0] >= 0.57735f) { // ~= 1 /
sqrt(3)
1363         manifold->tangent_vectors[0][0] = manifold-
>contact_normal[1];

```

```

1364         manifold->tangent_vectors[0][1] = -manifold-
>contact_normal[0];
1365         manifold->tangent_vectors[0][2] = 0.0f;
1366     } else {
1367         manifold->tangent_vectors[0][0] = 0.0f;
1368         manifold->tangent_vectors[0][1] = manifold-
>contact_normal[2];
1369         manifold->tangent_vectors[0][2] = -manifold-
>contact_normal[1];
1370     }
1371
1372     glm_vec3_normalize(manifold->tangent_vectors[0]);
1373     glm_vec3_cross(manifold->contact_normal, manifold-
>tangent_vectors[0], manifold->tangent_vectors[1]);
1374
1375     // zero out the impulses vector cuz junk was going in
1376     for (uint i = 0; i < NUM_CONSTRAINTS; i++) {
1377         manifold->impulses[i] = 0.0f;
1378     }
1379
1380     return SUCCESS;
1381 }
1382
1383 /**
1384 * Check for collisions between two arrays of triangles, and
copy collision data into an empty manifold.
1385 * @param triangles_a The first array of triangles.
1386 * @param num_triangles_a The number of triangles in the array
(number of points / 3)
1387 * @param triangles_b The second array of triangles.
1388 * @param num_triangles_b The number of triangles in the array
1389 * @param collision Set to 1 if there was a collision between
any of the triangles, 0 otherwise.
1390 * @param manifold A pointer to a manifold to potentially copy
data into.
1391 * @throws NOTHING but in case I ever change anything it might
throw something one day.
1392 */
1393 static exception tekCheckTrianglesCollision(vec3* triangles_a,
const uint num_triangles_a, vec3* triangles_b, const uint
num_triangles_b, flag* collision, TekCollisionManifold* manifold) {
1394     // get all possible combinations of triangles
1395     *collision = 0;

```

```

1396     for (uint i = 0; i < num_triangles_a; i++) {
1397         for (uint j = 0; j < num_triangles_b; j++) {
1398             flag sub_collision = 0;
1399             // issue - if there is more than one collision
then collision manifold gets overwritten.
1400             // i dont think that multiple triangles ever
happens though ngl.
1401
tekChainThrow(tekGetTriangleCollisionManifold(triangles_a + i * 3,
triangles_b + j * 3, &sub_collision, manifold));
1402             if (sub_collision)
1403                 *collision = 1;
1404             }
1405         }
1406     return SUCCESS;
1407 }
1408
1409 /**
1410  * Check if two coordinates are within a tiny tolerance of
each other. (less than 1e-6 units separating them)
1411  * @param point_a The first coordinate to check.
1412  * @param point_b The second coordinate to check.
1413  * @return 1 if they are very close, 0 otherwise.
1414 */
1415 static int tekIsCoordinateEquivalent(vec3 point_a, vec3
point_b) {
1416     vec3 delta;
1417     glm_vec3_sub(point_b, point_a, delta);
1418     // norm2 = square distance, to avoid square rooting
unnecessarily.
1419     return glm_vec3_norm2(delta) < EPSILON_SQUARED;
1420 }
1421
1422 /**
1423  * Check if two manifolds have a contact point that is within
a tiny tolerance of each other.
1424  * @param manifold_a The first manifold to check
1425  * @param manifold_b The second manifold to check
1426  * @return 1 if they are equivalent, 0 otherwise
1427 */
1428 static int tekIsManifoldEquivalent(TekCollisionManifold*
manifold_a, TekCollisionManifold* manifold_b) {
1429     // check that both points are equal

```

```

1430     return
1431         tekIsCoordinateEquivalent(manifold_a-
>contact_points[0], manifold_b->contact_points[0])
1432         && tekIsCoordinateEquivalent(manifold_a-
>contact_points[1], manifold_b->contact_points[1]);
1433 }
1434
1435 /**
1436 * Search the list of contact manifolds and see if a potential
new contact already exists.
1437 * @param manifold_vector The vector of manifolds
1438 * @param manifold The manifold to check against
1439 * @param contained A flag that is set to 1 if the manifold is
contained already.
1440 * @throws VECTOR_EXCEPTION possibly
1441 */
1442 static exception tekDoesManifoldContainContacts(const Vector*
manifold_vector, TekCollisionManifold* manifold, flag* contained) {
1443     *contained = 0;
1444     // linear search
1445     for (uint i = 0; i < manifold_vector->length; i++) {
1446         TekCollisionManifold* loop_manifold;
1447         tekChainThrow(vectorGetItemPtr(manifold_vector, i,
&loop_manifold));
1448         // if the bodies are different, dont care what the
coords say they have to be different contacts.
1449         if (loop_manifold->bodies[0] != manifold->bodies[0] ||
loop_manifold->bodies[1] != manifold->bodies[1])
1450             continue;
1451         // if coordinates are within a tiny tolerance of each
other, then reject this collision manifold
1452         if (tekIsManifoldEquivalent(manifold, loop_manifold))
{
1453             *contained = 1;
1454             return SUCCESS;
1455         }
1456     }
1457
1458     return SUCCESS;
1459 }
1460
1461 /**

```

```

1462  * Helper function to get a child, essentially a mapping of
1463  * (0,1) -> (node.left,node.right)
1463  */
1464 #define getChild(collider_node, i) (i == LEFT) ?
collider_node->data.node.left : collider_node->data.node.right
1465
1466 /**
1467  * Get the collision manifolds relating to the two bodies,
and add them to a provided vector. Gives information such as contact
position, depth, normals, tangent vectors etc.
1468  * @param body_a The first body involved in the potential
collision.
1469  * @param body_b The second body involved.
1470  * @param collision Flag that is set to 1 if there was a
collision, 0 if not.
1471  * @param manifold_vector The vector containing all the
manifolds that will be produced. Will not empty the vector, so the
same vector can be used to collect all the manifolds of an entire
colliding system / scenario.
1472  * @throws FAILURE if collider buffer not initialised.
1473  */
1474 exception tekGetCollisionManifolds(TekBody* body_a, TekBody*
body_b, flag* collision, Vector* manifold_vector) {
1475     // general process:
1476     // check for collision between each pair of sub-obb in the
colliding pair.
1477     // for each colliding pair, add that pair to the collider
stack.
1478     // if the child is a leaf node/triangle, then only
traverse the non-leaf node until two triangles are being checked.
1479     // if two triangles are found to collide, no need to keep
traversing the collider structure. can just create a manifold /
check for triangle-triangle collision.
1480
1481     // we need a collider buffer to store pairs of nodes. if
not initialised, cannot continue
1482     if (collider_init == DE_INITIALISED) return SUCCESS;
1483     if (collider_init == NOT_INITIALISED) tekThrow(FAILURE,
"Collider buffer was never initialised.");
1484
1485     // initial item to add to collider stack, body a and body
b's collider.
1486     *collision = 0;

```

```

1487     collider_buffer.length = 0;
1488     TekColliderNode* pair[2] = {
1489         body_a->collider, body_b->collider
1490     };
1491     tekUpdateOBB(&body_a->collider->obb, body_a->transform);
1492     tekUpdateOBB(&body_b->collider->obb, body_b->transform);
1493     tekChainThrow(vectorAddItem(&collider_buffer, &pair));
1494
1495     uint obb_obb_checks = 0;
1496     uint obb_triangle_checks = 0;
1497     uint triangle_triangle_checks = 0;
1498
1499     // tree traversal.
1500     while (vectorPopItem(&collider_buffer, &pair)) {
1501         if (pair[LEFT]->type == COLLIDER_NODE) {
1502             tekUpdateOBB(&pair[LEFT]->data.node.left->obb,
1503                         body_a->transform);
1504             tekUpdateOBB(&pair[LEFT]->data.node.right->obb,
1505                         body_a->transform);
1506         }
1507         if (pair[RIGHT]->type == COLLIDER_NODE) {
1508             tekUpdateOBB(&pair[RIGHT]->data.node.left->obb,
1509                         body_b->transform);
1510             tekUpdateOBB(&pair[RIGHT]->data.node.right->obb,
1511                         body_b->transform);
1512         }
1513         const uint i_max = pair[LEFT]->type == COLLIDER_NODE ?
1514             2 : 1;
1515         const uint j_max = pair[RIGHT]->type ==
1516             COLLIDER_NODE ? 2 : 1;
1517         // checking all possible pairs of children. could be
1518         // either 1, 2 or 4 checks.
1519         for (uint i = 0; i < i_max; i++) {
1520             for (uint j = 0; j < j_max; j++) {
1521                 // get child if it exists, else just the
1522                 // collider node.
1523                 TekColliderNode* node_a = (pair[LEFT]->type ==
1524                     COLLIDER_NODE) ? getChild(pair[LEFT], i) : pair[LEFT];

```

```

1520             TekColliderNode* node_b = (pair[RIGHT]->type
== COLLIDER_NODE) ? getChild(pair[RIGHT], j) : pair[RIGHT];
1521
1522             flag sub_collision = 0;
1523
1524             // use correct collision detection method
based on the types of colliders involved
1525             if ((node_a->type == COLLIDER_NODE) &&
(node_b->type == COLLIDER_NODE)) {
1526                 sub_collision =
tekCheckOBBCollision(&node_a->obb, &node_b->obb);
1527                 obb_obb_checks++;
1528             } else if ((node_a->type == COLLIDER_NODE) &&
(node_b->type == COLLIDER_LEAF)) {
1529                 tekUpdateLeaf(node_b, body_b->transform);
1530                 sub_collision =
tekCheckOBBTrianglesCollision(&node_a->obb, node_b-
>data.leaf.w_vertices, node_b->data.leaf.num_vertices / 3);
1531                 obb_triangle_checks++;
1532             } else if ((node_a->type == COLLIDER_LEAF) &&
(node_b->type == COLLIDER_NODE)) {
1533                 tekUpdateLeaf(node_a, body_a->transform);
1534                 sub_collision =
tekCheckOBBTrianglesCollision(&node_b->obb, node_a-
>data.leaf.w_vertices, node_a->data.leaf.num_vertices / 3);
1535                 obb_triangle_checks++;
1536             } else {
1537                 // triangle-triangle collision is more
special
1538                 // need to create a collision manifold if
there is a collision
1539                 tekUpdateLeaf(node_a, body_a->transform);
1540                 tekUpdateLeaf(node_b, body_b->transform);
1541                 TekCollisionManifold manifold;
1542                 tekChainThrow(tekCheckTrianglesCollision(
1543                     node_a->data.leaf.w_vertices, node_a-
>data.leaf.num_vertices / 3,
1544                     node_b->data.leaf.w_vertices, node_b-
>data.leaf.num_vertices / 3,
1545                     &sub_collision, &manifold
1546                     ));
1547
1548                 triangle_triangle_checks++;

```

```

1549
1550             if (sub_collision) {
1551                 sub_collision = 0;
1552                 manifold.bodies[0] = body_a;
1553                 manifold.bodies[1] = body_b;
1554
1555                 flag contained;
1556
tekChainThrow(tekDoesManifoldContainContacts(manifold_vector,
&manifold, &contained));
1557                     if (!contained &&
manifold.penetration_depth > EPSILON)
tekChainThrow(vectorAddItem(manifold_vector, &manifold));
1558                     *collision = 1;
1559                 }
1560             }
1561
1562             // if there was a triangle-triangle collision,
there has to be some form of collision between bodies, so mark as
such.
1563             if (sub_collision) {
1564                 temp_pair[LEFT] = node_a;
1565                 temp_pair[RIGHT] = node_b;
1566
tekChainThrow(vectorAddItem(&collider_buffer, &temp_pair));
1567             }
1568         }
1569     }
1570 }
1571
1572     const uint unoptimised = body_a->num_indices * body_b-
>num_indices / 9;
1573     printf("Un optimised: %u triangle-triangle checks.\n",
unoptimised);
1574     printf("Optimised : %u obb-obb checks.\n",
obb_obb_checks);
1575     printf("                 %u obb-triangle checks.\n",
obb_triangle_checks);
1576     printf("                 %u triangle-triangle checks.\n",
triangle_triangle_checks);
1577
1578     return SUCCESS;
1579 }
```

```

1580
1581  /**
1582   * Create an inverse mass matrix, kinda a 12x12 matrix, 0,1 =
body a inverse mass + inverse inertia tensor, 2,3 = body b ...
1583   * @param body_a The first body being collided.
1584   * @param body_b The second body being collected.
1585   * @param inv_mass_matrix[4] The outputted inverse inertia
matrix.
1586  */
1587  static void tekSetupInvMassMatrix(TekBody* body_a, TekBody*
body_b, mat3 inv_mass_matrix[4]) {
1588      // if the body is immovable, we can say it has an infinite
mass.
1589      // the inverse mass matrix is 1/infinity, which tends to 0
1590      // so we can just zero out both matrices.
1591      if (body_a->immovable) {
1592          glm_mat3_zero(inv_mass_matrix[0]);
1593          glm_mat3_zero(inv_mass_matrix[1]);
1594          // otherwise, calculate the actual matrices.
1595      } else {
1596          glm_mat3_identity(inv_mass_matrix[0]);
1597          glm_mat3_scale(inv_mass_matrix[0], 1.0f / body_a-
>mass);
1598          glm_mat3_copy(body_a->inverse_inertia_tensor,
inv_mass_matrix[1]);
1599      }
1600
1601      // repeat for body b
1602      if (body_b->immovable) {
1603          glm_mat3_zero(inv_mass_matrix[2]);
1604          glm_mat3_zero(inv_mass_matrix[3]);
1605      } else {
1606          glm_mat3_identity(inv_mass_matrix[2]);
1607          glm_mat3_scale(inv_mass_matrix[2], 1.0f / body_b-
>mass);
1608          glm_mat3_copy(body_b->inverse_inertia_tensor,
inv_mass_matrix[3]);
1609      }
1610  }
1611
1612  /**
1613   * Apply collision between two bodies, based on the contact
manifold between them.

```

```

1614     * @param body_a The first body that is colliding
1615     * @param body_b The second body that is colliding
1616     * @param manifold The contact manifold between the bodies
1617     * @throws SUCCESS nothing throws an exception.
1618 */
1619 exception tekApplyCollision(TekBody* body_a, TekBody* body_b,
TekCollisionManifold* manifold) {
1620     vec3 constraints[NUM_CONSTRAINTS][4];
1621     mat3 inv_mass_matrix[4];
1622     tekSetupInvMassMatrix(body_a, body_b, inv_mass_matrix);
1623
1624     const float friction = fmaxf(body_a->friction, body_b-
>friction);
1625
1626     // generate the impulse constraint
1627     glm_vec3_negate_to(manifold->contact_normal,
constraints[NORMAL_CONSTRAINT][0]);
1628     glm_vec3_cross(manifold->r_ac, manifold->contact_normal,
constraints[NORMAL_CONSTRAINT][1]);
1629     glm_vec3_negate(constraints[NORMAL_CONSTRAINT][1]);
1630     glm_vec3_copy(manifold->contact_normal,
constraints[NORMAL_CONSTRAINT][2]);
1631     glm_vec3_cross(manifold->r_bc, manifold->contact_normal,
constraints[NORMAL_CONSTRAINT][3]);
1632
1633     // generate the tangential / friction constraints
1634     for (uint j = 0; j < 2; j++) {
1635         glm_vec3_negate_to(manifold->tangent_vectors[j],
constraints[TANGENT_CONSTRAINT_1 + j][0]);
1636         glm_vec3_cross(manifold->r_ac, manifold-
>tangent_vectors[j], constraints[TANGENT_CONSTRAINT_1 + j][1]);
1637         glm_vec3_negate(constraints[TANGENT_CONSTRAINT_1 + j]
[1]);
1638         glm_vec3_copy(manifold->tangent_vectors[j],
constraints[TANGENT_CONSTRAINT_1 + j][2]);
1639         glm_vec3_cross(manifold->r_bc, manifold-
>tangent_vectors[j], constraints[TANGENT_CONSTRAINT_1 + j][3]);
1640     }
1641
1642     // for each constraint, solve to find change in velocity
1643     for (uint c = 0; c < NUM_CONSTRAINTS; c++) {
1644         float lambda_d = 0.0f; // denominator
1645         for (uint j = 0; j < 4; j++) {

```

```

1646             vec3 constraint;
1647             glm_mat3_mulv(inv_mass_matrix[j], constraints[c]
1648 [j], constraint);
1649             lambda_d += glm_vec3_dot(constraints[c][j],
constraint);
1650
1651             float lambda_n = 0.0f; // numerator
1652             lambda_n -= glm_vec3_dot(constraints[c][0], body_a-
>velocity);
1653             lambda_n -= glm_vec3_dot(constraints[c][1], body_a-
>angular_velocity);
1654             lambda_n -= glm_vec3_dot(constraints[c][2], body_b-
>velocity);
1655             lambda_n -= glm_vec3_dot(constraints[c][3], body_b-
>angular_velocity);
1656             if (c == NORMAL_CONSTRAINT) {
1657                 lambda_n -= manifold->baumgarte_stabilisation;
1658             }
1659
1660             float lambda = lambda_n / lambda_d;
1661
1662             // clamp lambda.
1663             const float temp = manifold->impulses[c];
1664             switch (c) {
1665                 case NORMAL_CONSTRAINT:
1666                     manifold->impulses[c] = fmaxf(manifold-
>impulses[c] + lambda, 0.0f);
1667                     lambda = manifold->impulses[c] - temp;
1668                     break;
1669                 case TANGENT_CONSTRAINT_1:
1670                 case TANGENT_CONSTRAINT_2:
1671                     manifold->impulses[c] = glm_clamp(manifold-
>impulses[c] + lambda, -friction * manifold-
>impulses[NORMAL_CONSTRAINT], friction * manifold-
>impulses[NORMAL_CONSTRAINT]);
1672                     lambda = manifold->impulses[c] - temp;
1673                     break;
1674                 default:
1675                     break;
1676             }
1677

```

```

1678         // calculate change in velocity for both bodies linear
and angular velocity.
1679         vec3 delta_v[4];
1680         for (uint j = 0; j < 4; j++) {
1681             glm_mat3_mulv(inv_mass_matrix[j], constraints[c]
[j], delta_v[j]);
1682             glm_vec3_scale(delta_v[j], lambda, delta_v[j]);
1683         }
1684
1685         // if body is immovable, then do not apply the change
1686         if (!body_a->immovable) {
1687             glm_vec3_add(body_a->velocity, delta_v[0], body_a-
>velocity);
1688             glm_vec3_add(body_a->angular_velocity, delta_v[1],
body_a->angular_velocity);
1689         }
1690         if (!body_b->immovable) {
1691             glm_vec3_add(body_b->velocity, delta_v[2], body_b-
>velocity);
1692             glm_vec3_add(body_b->angular_velocity, delta_v[3],
body_b->angular_velocity);
1693         }
1694     }
1695
1696     return SUCCESS;
1697 }
1698
1699 /**
1700 * Decide which bodies are colliding and apply impulses to
seperate any colliding bodies.
1701 * @param bodies The vector containing all the bodies.
1702 * @param phys_period The time period of the simulation.
1703 * @throws FAILURE if contact buffer was not initialised.
1704 */
1705 exception tekSolveCollisions(const Vector* bodies, const float
phys_period) {
1706     if (collider_init == DE_INITIALISED) return SUCCESS;
1707     if (collider_init == NOT_INITIALISED) tekThrow(FAILURE,
>Contact buffer was never initialised.");
1708
1709     contact_buffer.length = 0;
1710
1711     // loop through all pairs of bodies

```

```

1712     for (uint i = 0; i < bodies->length; i++) {
1713         TekBody* body_i;
1714         tekChainThrow(vectorGetItemPtr(bodies, i, &body_i));
1715         if (!body_i->num_vertices) continue;
1716         for (uint j = 0; j < i; j++) {
1717             TekBody* body_j;
1718             tekChainThrow(vectorGetItemPtr(bodies, j,
1719                                         &body_j));
1720             if (!body_j->num_vertices) continue;
1721             // if both immovable, they will be unaffected by
1722             // whatever response happens.
1723             if (body_i->immovable && body_j->immovable)
1724                 continue;
1725             // find contact points and add to the contact
1726             // buffer
1727             flag is_collision = 0;
1728             tekChainThrow(tekGetCollisionManifolds(body_i,
1729                                         body_j, &is_collision, &contact_buffer))
1730             }
1731         }
1732         // now loop through all contacts between bodies
1733         for (uint i = 0; i < contact_buffer.length; i++) {
1734             // get both bodies
1735             TekCollisionManifold* manifold;
1736             tekChainThrow(vectorGetItemPtr(&contact_buffer, i,
1737                                         &manifold));
1738             TekBody* body_a = manifold->bodies[0], * body_b =
1739             manifold->bodies[1];
1740             // get centres of both bodies
1741             // previously, i forgot that centre_of_mass != centre,
1742             // led to 24 (ish) hours of bug fixing LOL
1743             vec3 centre_a, centre_b;
1744             glm_vec3_add(body_a->position, body_a->centre_of_mass,
1745             centre_a);
1746             glm_vec3_add(body_b->position, body_b->centre_of_mass,
1747             centre_b);
1748             vec3 ab;

```

```

1745         glm_vec3_sub(centre_b, centre_a, ab);
1746
1747         // baumgarte stabilisation = stabilising force to
1748         prevent jitter
1749         manifold->baumgarte_stabilisation = -BAUMGARTE_BETA /
1750         phys_period * fmaxf(manifold->penetration_depth - SLOP, 0.0f);
1751         float restitution = fminf(body_a->restitution, body_b-
1752         >restitution);
1753
1754         // couple other quantites that are needed
1755         glm_vec3_sub(manifold->contact_points[0], centre_a,
1756         manifold->r_ac);
1757         glm_vec3_sub(manifold->contact_points[1], centre_b,
1758         manifold->r_bc);
1759
1760         vec3 delta_v;
1761         glm_vec3_sub(body_b->velocity, body_a->velocity,
1762         delta_v);
1763
1764         vec3 rw_a, rw_b;
1765         glm_vec3_cross(body_a->angular_velocity, manifold-
1766         >r_ac, rw_a);
1767         glm_vec3_cross(body_b->angular_velocity, manifold-
1768         >r_bc, rw_b);
1769         glm_vec3_negate(rw_a);
1770
1771         vec3 restitution_vector;
1772         glm_vec3_add(rw_a, rw_b, restitution_vector);
1773         glm_vec3_add(restitution_vector, delta_v,
1774         restitution_vector);
1775         restitution *= glm_vec3_dot(restitution_vector,
1776         manifold->contact_normal);
1777
1778         manifold->baumgarte_stabilisation += restitution;
1779     }
1780
1781     // iterative solver step -> repeat NUM_ITERATIONS time
1782     // through the iterations, the change in velocity to
1783     // separate approaches global solution
1784     for (uint s = 0; s < NUM_ITERATIONS; s++) {
1785         for (uint i = 0; i < contact_buffer.length; i++) {
1786             TekCollisionManifold* manifold;

```

```

1776             tekChainThrow(vectorGetItemPtr(&contact_buffer, i,
1777                                         &manifold));
1777             tekChainThrow(tekApplyCollision(manifold-
1778                                         >bodies[0], manifold->bodies[1], manifold));
1778         }
1779     }
1780
1781     return SUCCESS;
1782 }
```

tekphys/collisions.h

```

1 #pragma once
2
3 #include <cglm/vec3.h>
4
5 #include "../tekgl.h"
6 #include "../core/exception.h"
7 #include "../core/vector.h"
8 #include "body.h"
9
10 #define NORMAL_CONSTRAINT 0
11 #define TANGENT_CONSTRAINT_1 1
12 #define TANGENT_CONSTRAINT_2 2
13 #define NUM_CONSTRAINTS 3
14
15 #define NUM_ITERATIONS 32
16
17 #define BAUMGARTE_BETA 0.1f
18 #define MIN_PENETRATION 0.005f
19 #define SLOP 0.01f
20 #define POSITION_EPSILON 1e-4f
21
22 typedef struct TekCollisionManifold {
23     TekBody* bodies[2];
24     vec3 contact_points[2];
25     vec3 contact_normal;
26     vec3 tangent_vectors[2];
27     vec3 r_ac;
28     vec3 r_bc;
29     float penetration_depth;
30     float baumgarте_stabilisation;
31     float impulses[NUM_CONSTRAINTS];
32 } TekCollisionManifold;
```

```

33
34 int tekTriangleTest();
35 exception tekGetCollisionManifolds(TekBody* body_a, TekBody*
body_b, flag* collision, Vector* manifold_vector);
36 exception tekApplyCollision(TekBody* body_a, TekBody* body_b,
TekCollisionManifold* manifold);
37 exception tekSolveCollisions(const Vector* bodies, float
phys_period);

```

tekphys/engine.c

```

1 #include "engine.h"
2
3 #include <math.h>
4 #include <stdarg.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <time.h>
9 #include <sys/stat.h>
10 #include <math.h>
11 #include <pthread.h>
12 #include <cglm/euler.h>
13
14 #include "body.h"
15 #include "collider.h"
16 #include "../core/vector.h"
17 #include "../core/queue.h"
18 #include "GLFW/glfw3.h"
19 #include "collisions.h"
20
21 /**
22 * Template for generating receive functions for thread queues.
23 */
24 #define RECV_FUNC(func_name, func_type, param_name) \
25 exception func_name(ThreadQueue* queue, func_type* param_name) \
{ \
26     func_type* copy; \
27     if (!threadQueueDequeue(queue, &copy)) return FAILURE; \
28     memcpy(param_name, copy, sizeof(func_type)); \
29     free(copy); \
30     return SUCCESS; \
31 } \
32

```

```

33 /**
34  * @brief Recieve an event from the thread queue.
35  * @note Requires the event struct to exist already
36  * @param queue A pointer to the thread queue to recieve from.
37  * @param event A pointer to an empty TekEvent struct.
38  * @throws MEMORY_EXCEPTION if malloc() fails.
39 */
40 static RECV_FUNC(recvEvent, TekEvent, event);
41
42 /**
43  * @brief Receive a state from the thread queue.
44  * @note Requires the state struct to exist already.
45  * @param queue A pointer to the thread queue to recieve from.
46  * @param state A pointer to an empty TekState struct.
47  * @throws MEMORY_EXCEPTION if malloc() fails.
48 */
49 RECV_FUNC(recvState, TekState, state);
50
51 /**
52  * Template function for generating thread queue push
functions.
53 */
54 #define PUSH_FUNC(func_name, func_type, param_name) \
55 exception func_name(ThreadQueue* queue, const func_type
param_name) { \
56     func_type* copy = (func_type*)malloc(sizeof(func_type)); \
57     if (!copy) tekThrow(MEMORY_EXCEPTION, "Failed to allocate
memory for thread queue"); \
58     memcpy(copy, &param_name, sizeof(func_type)); \
59     if (!threadQueueEnqueue(queue, copy)) return FAILURE; \
60     return SUCCESS; \
61 } \
62
63 /**
64  * @brief Push a state to the thread queue.
65  * @param queue A pointer to the thread queue to push to.
66  * @param state A pointer to the state struct to push.
67  * @throws MEMORY_EXCEPTION if malloc() fails.
68 */
69 static PUSH_FUNC(pushState, TekState, state);
70
71 /**
72  * @brief Push an event to the thread queue.

```

```

73  * @param queue A pointer to the thread queue to push to.
74  * @param event A pointer to the event to push.
75  * @throws MEMORY_EXCEPTION if malloc() fails.
76  */
77 PUSH_FUNC(pushEvent, TekEvent, event);
78
79 /**
80  * Template for generating functions for updating the camera.
Actually obsolete now but too late to remove.
81 */
82 #define CAM_UPDATE_FUNC(func_name, param_name, state_type) \
83 static exception func_name(ThreadQueue* state_queue, const vec3 \
param_name) { \
84     TekState state = {}; \
85     memcpy(state.data.param_name, param_name, sizeof(vec3)); \
86     state.type = state_type; \
87     tekChainThrow(pushState(state_queue, state)); \
88     return SUCCESS; \
89 } \
90
91 /**
92  * @brief Send a print message through the state queue.
93  * @param state_queue The state queue to send the message
through.
94  * @param format The message to send + formatting e.g. "string:
%s"
95  * @param ... The formatting to add to the string
96  */
97 static void threadPrint(ThreadQueue* state_queue, const char*
format, ...) {
98     char temp_write;
99     va_list va;
100
101    va_start(va, format);
102    const int len_message = vsnprintf(&temp_write, 0, format,
va); // figure out the length of message
103    va_end(va);
104
105    char* buffer = (char*)malloc(len_message * sizeof(char));
// create buffer big enough
106    if (!buffer) return;
107
108    va_start(va, format);

```

```

109     vsprintf(buffer, format, va);
110     va_end(va);
111
112     // send message across
113     TekState message_state = {};
114     message_state.type = MESSAGE_STATE;
115     message_state.object_id = 0;
116     message_state.data.message = buffer;
117     pushState(state_queue, message_state);
118 }
119
120 /**
121 * Send print message through state queue. Acts like normal
122 printf()
123 */
124
125 /**
126 * @brief Send an exception message to the state queue.
127 * @param state_queue The state queue to send the exception to.
128 * @param exception The exception code.
129 */
130 static void threadExcept(ThreadQueue* state_queue, const uint
exception) {
131     // create state
132     TekState exception_state = {};
133     tekPrintException();
134
135     // fill with data
136     exception_state.type = EXCEPTION_STATE;
137     exception_state.object_id = 0;
138     exception_state.data.exception = exception;
139
140     // push to state queue
141     pushState(state_queue, exception_state);
142 }
143
144 #define tekEngineCreateBodyCleanup \
145     tekDeleteBody(&body); \
146     if (mesh_copy) free(mesh_copy); \
147     if (material_copy) free(material_copy) \
148

```

```

149 /**
150  * @brief Create a body given the mesh, material, position etc.
151  * @note Will create both a body and a corresponding entity on
152  *       the graphics thread. The object ids are assigned in order, filling
153  *       gaps in the order when they appear.
154  * @param state_queue The ThreadQueue that will be used to send
155  *       the entity creation message to the graphics thread.
156  * @param bodies A pointer to a vector that contains the
157  *       bodies.
158  * @param object_id The ID of the new body to create.
159  * @param mesh_filename The file that contains the mesh for the
160  *       object.
161  * @param material_filename The file that contains the material
162  *       for the object. Only used in the graphics thread.
163  * @param mass The mass of the body.
164  * @param friction The coefficient of friction.
165  * @param restitution The coefficient of restitution.
166  * @param position The position of the body in the world.
167  * @param rotation The rotation of the body as a quaternion.
168  * @param scale The scale in x, y, and z direction from the
169  *       original shape of the body.
170  * @throws MEMORY_EXCEPTION if malloc() fails.
171  */
172
173 static exception tekEngineCreateBody(ThreadQueue* state_queue,
174                                     Vector* bodies, const uint object_id, const char* mesh_filename,
175                                     const char* material_filename, const float mass, const float
176                                     friction, const float restitution, vec3 position, vec4 rotation,
177                                     vec3 scale) {
178
179     // create the body
180     TekBody body = {};
181     tekChainThrow(tekCreateBody(mesh_filename, mass, friction,
182                               restitution, position, rotation, &body));
183
184
185     // copy mesh and material files to new strings
186     const uint len_mesh = strlen(mesh_filename) + 1;
187     const uint len_material = strlen(material_filename) + 1;
188     char* mesh_copy = (char*)malloc(len_mesh);
189     char* material_copy = 0;
190     if (!mesh_copy) tekThrowThen(MEMORY_EXCEPTION, "Failed to
191     allocate memory to copy mesh filename.", {
192         tekEngineCreateBodyCleanup;
193     });
194     material_copy = (char*)malloc(len_material);

```

```

179      if (!material_copy) tekThrowThen(MEMORY_EXCEPTION, "Failed
to allocate memory to copy material filename.", {
180          tekEngineCreateBodyCleanup;
181      });
182      memcpy(mesh_copy, mesh_filename, len_mesh);
183      memcpy(material_copy, material_filename, len_material);
184
185      // create the state
186      TekState state = {};
187      state.object_id = object_id;
188
189      // if body id is greater than current length of the vector
190      // add a load of empty bodies before adding the real one
191      if (bodies->length <= object_id) {
192          TekBody dummy = {};
193          memset(&dummy, 0, sizeof(TekBody));
194          while (bodies->length < object_id) {
195              tekChainThrowThen(vectorAddItem(bodies, &dummy), {
196                  tekEngineCreateBodyCleanup;
197              });
198          }
199          tekChainThrowThen(vectorAddItem(bodies, &body), {
200              tekEngineCreateBodyCleanup;
201          });
202          // otherwise just add it straight in
203      } else {
204          TekBody* delete_body;
205          tekChainThrowThen(vectorGetItemPtr(bodies, object_id,
&delete_body), {
206              tekEngineCreateBodyCleanup;
207          });
208          if (delete_body->num_vertices != 0)
209              tekDeleteBody(delete_body);
210
211          tekChainThrowThen(vectorSetItem(bodies, object_id,
&body), {
212              tekEngineCreateBodyCleanup;
213          });
214      }
215
216      // finish the state and push it to the state queue
217      state.type = ENTITY_CREATE_STATE;
218      state.data.entity.mesh_filename = mesh_filename;

```

```

219     state.data.entity.material_filename = material_filename;
220     glm_vec3_copy(position, state.data.entity.position);
221     glm_vec4_copy(rotation, state.data.entity.rotation);
222     glm_vec3_copy(scale, state.data.entity.scale);
223
224     pushState(state_queue, state);
225
226     return SUCCESS;
227 }
228
229 /**
230  * @brief Update the position and rotation of a body given its
231  * object id.
232  * @param state_queue The ThreadQueue that links to the
233  * graphics thread.
234  * @param bodies A pointer to a vector containing the bodies.
235  * @param object_id The object id of the body to update.
236  * @param position The new position of the body.
237  * @param rotation The new rotation of the body as a
238  * quaternion.
239  * @param scale The new scale of the body.
240  * @throws ENGINE_EXCEPTION if the object id is invalid.
241  */
242 static exception tekEngineUpdateBody(ThreadQueue* state_queue,
243 const Vector* bodies, const uint object_id, vec3 position, vec4
244 rotation, vec3 scale) {
245     TekBody* body;
246     tekChainThrow(vectorGetItemPtr(bodies, object_id, &body));
247     if (body->num_vertices == 0) {
248         // if the number of vertices == 0, then body is not
249         // valid
250         // most likely, the whole thing is just zeroes
251         tekThrow(ENGINE_EXCEPTION, "Body ID is not valid.");
252     }
253
254     TekState state = {};
255     state.type = ENTITY_UPDATE_STATE;
256     state.object_id = object_id;
257     glm_vec3_copy(position, state.data.entity_update.position);
258     glm_vec4_copy(rotation, state.data.entity_update.rotation);
259     glm_vec3_copy(scale, state.data.entity_update.scale);
260
261     tekChainThrow(pushState(state_queue, state));

```

```

256     return SUCCESS;
257 }
258
259 /**
260  * @brief Delete a body, freeing the object id for reuse and
261  * removing the counterpart on the graphics thread.
262  * @param state_queue The ThreadQueue linking to the graphics
263  * thread.
264  * @param bodies A vector containing the bodies.
265  * @param object_id The id of the body to delete.
266  * @throws ENGINE_EXCEPTION if the object id is invalid.
267  */
268 static exception tekEngineDeleteBody(ThreadQueue* state_queue,
269 const Vector* bodies, const uint object_id) {
270     TekBody* body;
271     tekChainThrow(vectorGetItemPtr(bodies, object_id, &body));
272     if (body->num_vertices == 0) {
273         // if the number of vertices == 0, then body is not
274         // valid
275         // most likely, the whole thing is just zeroes
276         tekThrow(ENGINE_EXCEPTION, "Body ID is not valid.");
277     }
278
279     TekState state = {};
280     state.object_id = object_id;
281     state.type = ENTITY_DELETE_STATE;
282     tekChainThrow(pushState(state_queue, state));
283
284     // set the data stored to be zeroes
285     // removing the body would change the index of other bodies
286     // index needed to find object by id.
287     tekDeleteBody(body);
288     memset(body, 0, sizeof(TekBody));
289     return SUCCESS;
290 }
291
292 /**
293  * Delete all non null bodies in the bodies vector.
294  * @param state_queue The thread queue of the simulation that
295  * sends states.
296  * @param bodies The vector containing all bodies in the
297  * simulation.
298  * @throws VECTOR_EXCEPTION .

```

```

293     */
294 static exception tekEngineDeleteAllBodies(ThreadQueue*
state_queue, const Vector* bodies) {
295     // loop over bodies
296     for (uint i = 0; i < bodies->length; i++) {
297         TekBody* body;
298         tekChainThrow(vectorGetItemPtr(bodies, i, &body));
299         if (body->num_vertices == 0) continue; // num
vertices==0 = no body
300
301         tekChainThrow(tekEngineDeleteBody(state_queue, bodies,
i));
302     }
303     return SUCCESS;
304 }
305
306 /**
307 * Push an inspect state to the state queue. This gives the
information about the body currently being inspected by the debug
menu.
308 * @param state_queue The thread queue to push the state to.
309 * @param time The current simulation time.
310 * @param position The current position of the body.
311 * @param velocity The current rotation of the body.
312 * @throws MEMORY_EXCEPTION if malloc() fails.
313 */
314 static exception tekPushInspectState(ThreadQueue* state_queue,
const float time, vec3 position, vec3 velocity) {
315     // create a new empty state
316     TekState state = {};
317
318     // write data
319     state.type = INSPECT_STATE;
320     state.data.inspect.time = time;
321     glm_vec3_copy(position, state.data.inspect.position);
322     glm_vec3_copy(velocity, state.data.inspect.velocity);
323
324     // push to state queue
325     tekChainThrow(pushState(state_queue, state));
326
327     return SUCCESS;
328 }
329

```

```

330  /// Store the args to link physics thread to graphics thread,
so it can be passed as a single pointer to the thread procedure.
331  struct TekEngineArgs {
332      ThreadQueue* event_queue;
333      ThreadQueue* state_queue;
334      double phys_period;
335  };
336
337 /**
338  * Call exception, push exception to state queue and goto
cleanup
339 */
340 #define threadThrow(exception_code, exception_message) { const
exception __thread_exception = exception_code; if
(__thread_exception) { tekSetException(__thread_exception, __LINE__,
__FUNCTION__, __FILE__, exception_message);
threadExcept(state_queue, __thread_exception); goto
tek_engine_cleanup; } }
341 /**
342  * Call an exception, push to state queue and goto cleanup
343 */
344 #define threadChainThrow(exception_code) { const exception
__thread_exception = exception_code; if (__thread_exception)
{ tekTraceException(__thread_exception, __LINE__, __FUNCTION__,
__FILE__); threadExcept(state_queue, __thread_exception); goto
tek_engine_cleanup; } }
345
346 /**
347  * @brief The main physics thread procedure, will run in
parallel to the graphics thread. Responsible for logic, has a loop
running at fixed time interval.
348  * @param args Required for use with pthread library. Used to
pass TekEngineArgs pointer.
349 */
350 static void tekEngine(void* args) {
351     // expand out engine args
352     const struct TekEngineArgs* engine_args = (struct
TekEngineArgs*)args;
353     ThreadQueue* event_queue = engine_args->event_queue;
354     ThreadQueue* state_queue = engine_args->state_queue;
355     double phys_period = engine_args->phys_period;
356     free(args);
357

```

```

358     // vector to store all the bodies being simulated
359     Vector bodies = {};
360     threadChainThrow(vectorCreate(0, sizeof(TekBody),
361 &bodies));
361
362     Queue unused_ids = {};
363     queueCreate(&unused_ids);
364
365     // variables used in the physics loop
366     struct timespec engine_time, step_time;
367     double phys_period_s = floor(phys_period);
368     step_time.tv_sec = (__time_t)phys_period_s;
369     step_time.tv_nsec = (__syscall_slong_t)(1e9 * (phys_period
- phys_period_s));
370     clock_gettime(CLOCK_MONOTONIC, &engine_time);
371     flag running = 1;
372     uint counter = 0;
373     flag mode = 0;
374     flag paused = 0;
375     flag step = 0;
376     float gravity = 9.81f;
377     uint inspect_index = 0;
378     float time_elapsed = 0.0f;
379
380     mat4 snapshot_rotation_matrix;
381     vec4 snapshot_rotation_quat;
382     TekBody* snapshot_body;
383
384     // main loop
385     while (running) {
386         // receive all events from window thread
387         TekEvent event = {};
388         while (recvEvent(event_queue, &event) == SUCCESS) {
389             // switch event type, each type has corresponding
data and action to be performed
390             switch (event.type) {
391                 case QUIT_EVENT:
392                     // end the loop
393                     running = 0;
394
395                     // free the thread queues, as they won't be
needed any more
396                     threadQueueDelete(event_queue);

```

```

397         threadQueueDelete(state_queue);
398         break;
399     case MODE_CHANGE_EVENT:
400         mode = event.data.mode;
401         if (mode == MODE_RUNNER)
402             time_elapsed = 0.0f;
403         break;
404     case BODY_CREATE_EVENT:
405         // cannot convert directly for some reason
406         glm_euler(event.data.body.snapshot.rotation,
snapshot_rotation_matrix);
407         glm_mat4_quat(snapshot_rotation_matrix,
snapshot_rotation_quat);
408
409         threadChainThrow(tekEngineCreateBody(
410             state_queue, &bodies, event.data.body.id,
411             event.data.body.snapshot.model,
event.data.body.snapshot.material,
412             event.data.body.snapshot.mass,
event.data.body.snapshot.friction,
event.data.body.snapshot.restitution,
413             event.data.body.snapshot.position,
snapshot_rotation_quat, (vec3){1.0f, 1.0f, 1.0f}
414         ));
415
416         threadChainThrow(vectorGetItemPtr(&bodies,
event.data.body.id, &snapshot_body));
417
glm_vec3_copy(event.data.body.snapshot.velocity, snapshot_body-
>velocity);
418         snapshot_body->immovable =
event.data.body.snapshot.immovable;
419
420         break;
421     case BODY_UPDATE_EVENT:
422         // cannot convert directly for some reason
423         glm_euler(event.data.body.snapshot.rotation,
snapshot_rotation_matrix);
424         glm_mat4_quat(snapshot_rotation_matrix,
snapshot_rotation_quat);
425
glm_euler_xyz_quat_rh(event.data.body.snapshot.rotation,
snapshot_rotation_quat);

```

```

426
427             threadChainThrow(vectorGetItemPtr(&bodies,
event.data.body.id, &snapshot_body));
428
429             glm_vec3_copy(event.data.body.snapshot.position, snapshot_body-
>position);
430             glm_vec4_copy(snapshot_rotation_quat,
snapshot_body->rotation);
431
432             glm_vec3_copy(event.data.body.snapshot.angular_velocity,
snapshot_body->angular_velocity);
433             snapshot_body->friction =
event.data.body.snapshot.friction;
434             snapshot_body->restitution =
event.data.body.snapshot.restitution;
435             threadChainThrow(tekBodySetMass(snapshot_body,
event.data.body.snapshot.mass));
436             snapshot_body->immovable =
event.data.body.snapshot.immovable;
437             threadChainThrow(tekEngineUpdateBody(
438                     state_queue, &bodies, event.data.body.id,
439                     event.data.body.snapshot.position,
snapshot_rotation_quat, (vec3){1.0f, 1.0f, 1.0f}
440                 ));
441
442             break;
443         case BODY_DELETE_EVENT:
444
threadChainThrow(tekEngineDeleteBody(state_queue, &bodies,
event.data.body.id));
445             break;
446         case CLEAR_EVENT:
447
threadChainThrow(tekEngineDeleteAllBodies(state_queue, &bodies));
448             break;
449         case TIME_EVENT: // update physics time step
450             phys_period = 1 / event.data.time.rate;
451             phys_period_s = floor(phys_period /
event.data.time.speed);

```

```

452             step_time.tv_sec = (__time_t)(phys_period_s /
event.data.time.speed);
453             step_time.tv_nsec = (__syscall_slong_t)(1e9 *
(phys_period / event.data.time.speed - phys_period_s));
454             break;
455         case PAUSE_EVENT:
456             paused = event.data.paused;
457             break;
458         case STEP_EVENT: // advance by one simulation step
459             if (paused)
460                 step = 1;
461             break;
462         case GRAVITY_EVENT: // update acceleration due to
gravity
463             gravity = event.data.gravity;
464         case INSPECT_EVENT: // change which body is being
inspected
465             inspect_index = (uint)event.data.body.id;
466         default:
467             break;
468         }
469     }
470
471     if (!running) break;
472
473     if (step) paused = 0;
474
475     // sort out collisions
476     if (mode == MODE_RUNNER && !paused) {
477         // check and fix collisions
478         threadChainThrow(tekSolveCollisions(&bodies,
(float)phys_period));
479
480         for (uint i = 0; i < bodies.length; i++) {
481             TekBody* body = 0;
482             threadChainThrow(vectorGetItemPtr(&bodies, i,
&body));
483
484             // dont simulate null bodies
485             if (!body->num_vertices) continue;
486
487             // dont move immovable bodies
488             if (body->immovable) {

```

```

489             glm_vec3_zero(body->velocity);
490             glm_vec3_zero(body->angular_velocity);
491         }
492         tekBodyAdvanceTime(body, (float)phys_period,
493 gravity);
493
494     threadChainThrow(tekEngineUpdateBody(state_queue, &bodies, i, body-
495 >position, body->rotation, body->scale));
494     }
495
496     time_elapsed += phys_period;
497 }
498
499 // allows for the sim to be stepped one step at a time
500 if (step) {
501     paused = 1;
502     step = 0;
503 }
504
505 // return details about a specific body to debug
506 vec3 inspect_position, inspect_velocity;
507 if (inspect_index < bodies.length) {
508     TekBody* inspect_body = 0;
509     threadChainThrow(vectorGetItemPtr(&bodies,
510 inspect_index, &inspect_body));
510     glm_vec3_copy(inspect_body->position,
511 inspect_position);
511     glm_vec3_copy(inspect_body->velocity,
512 inspect_velocity);
512 } else {
513     glm_vec3_zero(inspect_position);
514     glm_vec3_zero(inspect_velocity);
515 }
516 threadChainThrow(tekPushInspectState(state_queue,
517 time_elapsed, inspect_position, inspect_velocity));
518
519 // update engine time
520 engine_time.tv_sec += step_time.tv_sec;
521 engine_time.tv_nsec += step_time.tv_nsec;
522
523 // wait for time to pass before moving onto next step
524 while (engine_time.tv_nsec >= BILLION) {
524     engine_time.tv_nsec -= BILLION;

```

```

525             engine_time.tv_sec += 1;
526         }
527
528         clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
&engine_time, NULL);
529         counter++;
530     }
531
532     // goto here after the loop finished or terminates
unexpectedly
533 tek_engine_cleanup:
534     for (uint i = 0; i < bodies.length; i++) {
535         TekBody* loop_body;
536         threadChainThrow(vectorGetItemPtr(&bodies, i,
&loop_body));
537         tekDeleteBody(loop_body);
538     }
539
540     vectorDelete(&bodies);
541     queueDelete(&unused_ids);
542 }
543
544 /**
545 * @brief Start the physics thread, providing two thread queues
to send and receive events / states.
546 * @param event_queue A pointer to an existing thread queue
that will send events to the physics thread.
547 * @param state_queue A pointer to an existing thread queue
that will receive updates of the physics state.
548 * @param phys_period A time period that represents the length
of a single iteration of the physics loop. In other words '1 / ticks
per second'
549 * @param thread A pointer to a pthread_t variable that will
contain the thread. Do pthread_join(thread) at the end to ensure the
thread is finished.
550 * @throws THREAD_EXCEPTION if the call to pthread_create()
fails.
551 */
552 exception tekInitEngine(ThreadQueue* event_queue, ThreadQueue*
state_queue, const double phys_period, unsigned long long* thread) {
553     // engine args passed into thread. but can only pass one
pointer in hence the struct.

```

```

554     struct TekEngineArgs* engine_args = (struct
555         TekEngineArgs*)malloc(sizeof(struct TekEngineArgs));
556     engine_args->event_queue = event_queue;
557     engine_args->state_queue = state_queue;
558     engine_args->phys_period = phys_period;
559     // create new thread, if returns not 0 then there was a
560     // bugger
561     if (pthread_create((pthread_t*)thread, NULL, tekEngine,
562                         engine_args))
563         tekThrow(THREAD_EXCEPTION, "Failed to create physics
564                         thread");
565     return SUCCESS;
566 }
567 /**
568 * Wait for the engine thread to join.
569 * @param thread The engine thread, returned by tekInitEngine()
570 */
571 void tekAwaitEngineStop(const unsigned long long thread) {
572     // wrapper so u dont have to include pthread.h or whatnot
573     pthread_join(thread, NULL);
574 }
```

tekphys/engine.h

```

1 #pragma once
2
3 #include "../core/exception.h"
4 #include "../core/threadqueue.h"
5
6 #include "../tekgl/entity.h"
7 #include "body.h"
8
9 #define QUIT_EVENT      0
10 #define MODE_CHANGE_EVENT 1
11 #define BODY_CREATE_EVENT 2
12 #define BODY_DELETE_EVENT 3
13 #define BODY_UPDATE_EVENT 4
14 #define CLEAR_EVENT    5
15 #define TIME_EVENT      6
16 #define PAUSE_EVENT     7
17 #define STEP_EVENT      8
18 #define GRAVITY_EVENT   9
19 #define INSPECT_EVENT   10
```

```

20
21 #define MESSAGE_STATE      0
22 #define EXCEPTION_STATE   1
23 #define ENTITY_CREATE_STATE 2
24 #define ENTITY_DELETE_STATE 3
25 #define ENTITY_UPDATE_STATE 4
26 #define INSPECT_STATE      5
27
28 typedef struct TekEvent {
29     flag type;
30     union {
31         struct {
32             TekBodySnapshot snapshot;
33             uint id;
34         } body;
35         flag mode;
36         struct {
37             double rate;
38             double speed;
39         } time;
40         flag paused;
41         float gravity;
42     } data;
43 } TekEvent;
44
45 typedef struct TekState {
46     flag type;
47     uint object_id;
48     union {
49         char* message;
50         uint exception;
51         struct {
52             const char* mesh_filename;
53             const char* material_filename;
54             vec3 position;
55             vec4 rotation;
56             vec3 scale;
57         } entity;
58         struct {
59             vec3 position;
60             vec4 rotation;
61             vec3 scale;
62         } entity_update;

```

```

63         struct {
64             float time;
65             vec3 position;
66             vec3 velocity;
67         } inspect;
68     } data;
69 } TekState;
70
71 exception recvState(ThreadQueue* queue, TekState* state);
72 exception pushEvent(ThreadQueue* queue, TekEvent event);
73 exception tekInitEngine(ThreadQueue* event_queue, ThreadQueue* state_queue, double phys_period, unsigned long long* thread);
74 void tekAwaitEngineStop(unsigned long long thread);
75
76 exception pushTriangle(vec3 triangle[3]);
77 exception push0BB(void* obb);

```

tekphys/geometry.c

```

1 #include "geometry.h"
2
3 #include "../tekgl.h"
4 #include <cglm/vec4.h>
5 #include <stdarg.h>
6 #include <time.h>
7 #include <cglm/mat3.h>
8 #include <cglm/mat4.h>
9 #include "../tekgl/manager.h"
10
11 /**
12  * Called when program begins, initialise some things for
13  * geometry.
14 */
15 tek_init tekInitGeometry() {
16     // seed random number generator. (not cryptographically
17     // secure but that doesn't matter here)
18     srand(time(0));
19 }
20 /**
21  * @brief Find the sum of a number of vec3s.
22  * @note Stops counting when a null pointer is reached. Doesn't
23  * check if values are actually vec3s.
24  * @param dest Where to store the sum.

```

```

23  * @param ... Vec3s to sum.
24  */
25 void sumVec3VA(vec3 dest, ...) {
26     // using variadic arguments to allow for any number of
vectors
27     // macro makes the final argument be nullptr
28     va_list vec3_list;
29     va_start(vec3_list, dest);
30     float* vector_ptr;
31     glm_vec3_zero(dest);
32     while ((vector_ptr = va_arg(vec3_list, float*))) {
33         glm_vec3_add(vector_ptr, dest, dest);
34     }
35 }
36
37 /**
38  * @brief Calculate the signed volume of a tetrahedron from its
four vertices.
39  *
40  * @param point_a Vertex A of tetrahedron
41  * @param point_b Vertex B of tetrahedron
42  * @param point_c Vertex C of tetrahedron
43  * @param point_d Vertex D of tetrahedron
44  * @return The signed volume of the tetrahedron (e.g. can be
either positive or negative volume)
45 */
46 float tetrahedronSignedVolume(const vec3 point_a, const vec3
point_b, const vec3 point_c, const vec3 point_d) {
47     // volume of a tetrahedron can be computed using a matrix
48     // https://en.wikipedia.org/wiki/Tetrahedron#Volume
49
50     mat4 matrix;
51     for (uint i = 0; i < 3; i++) {
52         matrix[0][i] = point_a[i];
53         matrix[1][i] = point_b[i];
54         matrix[2][i] = point_c[i];
55         matrix[3][i] = point_d[i];
56     }
57     for (uint i = 0; i < 4; i++) {
58         matrix[i][3] = 1.0f;
59     }
60     const float determinant = glm_mat4_det(matrix);
61     return determinant / 6.0f;

```

```

62 }
63 /**
64 * @brief Find the outer product of two vectors, a⊗b
65 * @param a Left-hand vector
66 * @param b Right-hand vector
67 * @param m Resultant matrix
68 */
69
70 static void mat3OuterProduct(const vec3 a, const vec3 b, mat3
m) {
71     // outer product is just a matrix that has all pairs of
multiplications between the vectors.
72     for (uint i = 0; i < 3; i++) {
73         for (uint j = 0; j < 3; j++) {
74             m[i][j] = a[j] * b[i];
75         }
76     }
77 }
78
79 /**
80 * @brief Find the sum of two matrices.
81 *
82 * @note Acceptable for a or b to be the same matrix as m.
83 *
84 * @param a Left-hand matrix
85 * @param b Right-hand matrix
86 * @param m Result
87 */
88 void mat3Add(mat3 a, mat3 b, mat3 m) {
89     // 3 tall, 3 wide, so need a nested loop
90     for (uint i = 0; i < 3; i++) {
91         for (uint j = 0; j < 3; j++) {
92             m[i][j] = a[i][j] + b[i][j];
93         }
94     }
95 }
96
97 /**
98 * @brief Find the difference between two matrices.
99 *
100 * @note Acceptable for a or b to be the same matrix as m.
101 *
102 * @param a Left-hand matrix

```

```

103 * @param b Right-hand matrix
104 * @param m Result
105 */
106 static void mat3Subtract(mat3 a, mat3 b, mat3 m) {
107     // 3 tall, 3 wide, so need nested loop
108     for (uint i = 0; i < 3; i++) {
109         for (uint j = 0; j < 3; j++) {
110             m[i][j] = a[i][j] - b[i][j];
111         }
112     }
113 }
114
115 /**
116 * Calculate the moment of inertia of a tetrahedron, these make
117 up the leading diagonal of a inertia tensor matrix.
118 * @param a The set of coordinates in a single axis of each
119 point (e.g. all x-coordinates)
120 * @param b The matching coordinates in another axis (e.g. all
121 y-coordinates)
122 * @param mass The mass of the tetrahedron.
123 * @return The moment of inertia.
124 */
125 static float momentOfInertia(const vec4 a, const vec4 b, const
126 float mass) {
127     // calculating moments of inertia e.g. values of inertia
128 tensor along diagonal
129
130     // formula found here:
https://docsdrive.com/pdfs/sciencepublications/jmssp/2005/8-11.pdf
131     // "Explicit and exact formulas for the 3-D tetrahedron
132 inertia tensor in terms of its vertex coordinates"
133
134     // density * det(jacobian) * (a12 + a1a2 + a22 + a1a3 +
135 a2a3 + a32 + a1a4 + a2a4 + a3a4 + a42 + b12 + b1b2 + b22 + b1b3 +
136 b2b3 + b32 + b1b4 + b2b4 + b3b4 + b42) / 60
137     // det(jacobian) = 6 * volume
138     // density * 6 * volume * (...) / 60
139     // mass * (...) / 10
140     const float a1 = a[0], a2 = a[1], a3 = a[2], a4 = a[3];
141     const float b1 = b[0], b2 = b[1], b3 = b[2], b4 = b[3];
142     return 0.1f * mass * (a1 * a1 + a1 * a2 + a2 * a2 + a1 * a3
143 +

```

```

135             a2 * a3 + a3 * a3 + a1 * a4 + a2 * a4
+
136             a3 * a4 + a4 * a4 + b1 * b1 + b1 * b2
+
137             b2 * b2 + b1 * b3 + b2 * b3 + b3 * b3
+
138             b1 * b4 + b2 * b4 + b3 * b4 + b4 *
b4);
139 }
140
141 /**
142  * Helper function to calculate "products of inertia" - used as
elements in the inertia tensor.
143  * @param a A component of each point on a tetrahedron in one
axis. (e.g. all x-coordinates).
144  * @param b The matching components of a different axis. (e.g.
all y-coordinates)
145  * @param mass The mass of the tetrahedron.
146  * @return The "product of inertia"
147 */
148 static float productOfInertia(const vec4 a, const vec4 b, const
float mass) {
149     // calculating products of inertia e.g. values of inertia
not in the diagonal
150
151     // formula found here:
https://docsdrive.com/pdfs/sciencepublications/jmssp/2005/8-11.pdf
152     // "Explicit and exact formulas for the 3-D tetrahedron
inertia tensor in terms of its vertex coordinates"
153
154     // density * det(jacobian) * (2a1b1 + a2b1 + a3b1 + a4b1 +
a1b2 + 2a2b2 + a3b2 + a4b2 + a1b3 + a2b3 + 2a3b3 + a4b3 + a1b4 +
a2b4 + a3b4 + 2a4b4) / 120
155     // density * 6 * volume * (...) / 120
156     // mass * (...) / 20
157     const float a1 = a[0], a2 = a[1], a3 = a[2], a4 = a[3];
158     const float b1 = b[0], b2 = b[1], b3 = b[2], b4 = b[3];
159     return 0.05f * mass * (2.0f * a1 * b1 +
a2 * b1 +
a3 * b1 + a4 * b1 +
a1 * b2 + 2.0f * a2 * b2 +
a3 * b2 + a4 * b2 +
a1 * b3 + a2 * b3 +
2.0f * a3 * b3 + a4 * b3 +

```

```

162                               a1 * b4 +           a2 * b4 +
a3 * b4 + 2.0f * a4 * b4);
163 }
164
165 /**
166  * Calculate the inertia tensor of a tetrahedron based on the
167 four points that make it up.
168  * @param point_a The first point of the tetrahedron.
169  * @param point_b The second point of the tetrahedron.
170  * @param point_c The third point of the tetrahedron.
171  * @param point_d The fourth point of the tetrahedron.
172  * @param mass The mass of the tetrahedron.
173  * @param tensor The outputted inertia tensor.
174 */
175 void tetrahedronInertiaTensor(const vec3 point_a, const vec3
point_b, const vec3 point_c, const vec3 point_d, const float mass,
mat3 tensor) {
176     // calculating the inertia tensor
177
178     // formula found here:
https://docsdrive.com/pdfs/sciencepublications/jmssp/2005/8-11.pdf
179     // "Explicit and exact formulas for the 3-D tetrahedron
inertia tensor in terms of its vertex coordinates"
180
181     const vec4 x_values = { point_a[0], point_b[0], point_c[0],
point_d[0] };
182     const vec4 y_values = { point_a[1], point_b[1], point_c[1],
point_d[1] };
183     const vec4 z_values = { point_a[2], point_b[2], point_c[2],
point_d[2] };
184
185     // calculating moments of inertia in all 3 axes
186     const float am = momentOfInertia(y_values, z_values, mass);
187     const float bm = momentOfInertia(x_values, z_values, mass);
188     const float cm = momentOfInertia(x_values, y_values, mass);
189
190     // calculating products of inertia
191     const float ap = -productOfInertia(y_values, z_values,
mass);
192     const float bp = -productOfInertia(x_values, z_values,
mass);

```

```

193     const float cp = -productOfInertia(x_values, y_values,
mass);
194
195     // as stated in resource linked above:
196     //
197     //          |  a   -b'  -c' |
198     // Tensor = | -b'   b   -a' |
199     //          | -c'  -a'   c  |
200     //
201     // where a, b, c are moments of inertia
202     //      a', b', c' are products of inertia
203
204     mat3 tensor_temp = {
205         { am, bp, cp },
206         { bp, bm, ap },
207         { cp, ap, cm }
208     };
209
210     glm_mat3_copy(tensor_temp, tensor);
211 }
212
213 /**
214 * Translate an inertia tensor.
215 * @param tensor The inertia tensor to translate.
216 * @param mass The mass being added to the object.
217 * @param translate The displacement of the new mass / how far
to translate the tensor.
218 */
219 void translateInertiaTensor(mat3 tensor, const float mass, vec3
translate) {
220     // translating inertia tensor
221     // using formula from:
https://en.wikipedia.org/wiki/Parallel\_axis\_theorem#Tensor\_generalization
222     //
223     // Translated Inertia Tensor = It + m[(R·R)I - R ⊗ R]
224     //
225     // where It is the original tensor, R is the vector from
the new centre to the old centre, m is the mass, I is the identity
matrix
226
227     // calculating R·R

```

```

228     const float correction_scalar = glm_vec3_dot(translate,
229     translate);
230
231     mat3 tensor_mod;
232
233     // calculating (R·R)I
234     glm_mat3_identity(tensor_mod);
235     glm_mat3_scale(tensor_mod, correction_scalar);
236
237     // calculating R ⊗ R
238     mat3 outer_product;
239     mat3OuterProduct(translate, translate, outer_product);
240
241     // calculating (R·R)I - R ⊗ R
242     mat3Subtract(tensor_mod, outer_product, tensor_mod);
243
244     // calculating m[(R·R)I - R ⊗ R]
245     glm_mat3_scale(tensor_mod, mass);
246
247     // calculating I_t + m[(R·R)I - R ⊗ R]
248     mat3Add(tensor, tensor_mod, tensor);
249 }
250
251 /**
252 * Calculate the scalar triple product of three vectors
253 * = cross product of b and c, dotted against a.
254 * @param vector_a The first vector involved in the product
255 * @param vector_b The second vector involved in the product
256 * @param vector_c The third vector involved in the product
257 * @return A floating point number, which is the scalar triple
258 * product.
259 */
260 float scalarTripleProduct(vec3 vector_a, vec3 vector_b, vec3
vector_c) {
261     vec3 cross_product;
262     // first: b × c
263     glm_vec3_cross(vector_b, vector_c, cross_product);
264     // finally: (b × c) . a
265     return glm_vec3_dot(vector_a, cross_product);
266 }
267
268 /**
269 * Get the normal vector given the three points of a triangle.

```

```

267  * @param[in] triangle[3] An array of 3 vec3s representing the
triangle.
268  * @param[out] normal The normalised normal vector of this
triangle.
269  */
270 void triangleNormal(vec3 triangle[3], vec3 normal) {
271     // triangle normal = cross product of two of its edges.
272     // cross product finds perpendicular vector, so if both
edges are on the same plane, must get the normal
273     vec3 vec_ab, vec_ac;
274     glm_vec3_sub(triangle[1], triangle[0], vec_ab);
275     glm_vec3_sub(triangle[2], triangle[0], vec_ac);
276     glm_vec3_cross(vec_ab, vec_ac, normal);
277
278     // normalize for consistent results.
279     glm_vec3_normalize(normal);
280 }
281
282 /**
283  * Generate a random floating point number within two bounds,
min and max.
284  * @param min The minimum possible number to be generated.
285  * @param max The maximum possible number to be generated.
286  * @return A random number in between min and max.
287 */
288 float randomFloat(const float min, const float max) {
289     // rannd() = random *integer* between 0 and RAND_MAX.
290     // divide by RAND_MAX to get between 0.0 and 1.0
291     // multiply by range and add min, to scale to min, max
292     return ((max - min) * ((float)rand() / (float)RAND_MAX)) +
min;
293 }
```

tekphys/geometry.h

```

1 #pragma once
2
3 #include <cglm/vec3.h>
4
5 void sumVec3VA(vec3 dest, ...);
6
7 /// @copydoc sumVec3VA
8 #define sumVec3(dest, ...) sumVec3VA(dest, __VA_ARGS__, 0);
9
```

```

10 float tetrahedronSignedVolume(const vec3 point_a, const vec3
point_b, const vec3 point_c, const vec3 point_d);
11 void tetrahedronInertiaTensor(const vec3 point_a, const vec3
point_b, const vec3 point_c, const vec3 point_d, float mass, mat3
tensor);
12 void translateInertiaTensor(mat3 tensor, float mass, vec3
translate);
13
14 void mat3Add(mat3 a, mat3 b, mat3 m);
15 float scalarTripleProduct(vec3 vector_a, vec3 vector_b, vec3
vector_c);
16 void triangleNormal(vec3 triangle[3], vec3 normal);
17 float randomFloat(float min, float max);

```

tekphys/scenario.c

```

1 #include "scenario.h"
2
3 #include <stdio.h>
4 #include <string.h>
5 #include <linux/limits.h>
6
7 #include "../core/file.h"
8
9 #define SNAPSHOT_WRITE_FORMAT "ID:%u\nNAME:%s\nPOSITION:%f %f
%f\nROTATION:%f %f %f %f\nVELOCITY:%f %f %f\nMASS:%f\nCOEF_FRICTION:
%f\nCOEF_RESTITUTION:%f\nIMMOVABLE:%d\nMODEL:%s\nMATERIAL:%s\n"
10 #define SNAPSHOT_READ_FORMAT "ID:%u\nNAME:[^\n]\nPOSITION:%f
%f %f\nROTATION:%f %f %f %f\nVELOCITY:%f %f %f\nMASS:%f\
nCOEF_FRICTION:%f\nCOEF_RESTITUTION:%f\nIMMOVABLE:%d\nMODEL:[^\n]\\
nMATERIAL:[^\n]\n"
11 #define SNAPSHOT_NUM_LINES 11
12
13 struct TekScenarioPair {
14     TekBodySnapshot* snapshot;
15     ListItem* list_ptr;
16     uint id;
17 };
18
19 /**
20 * Get a snapshot from a scenario using its id.
21 * @param scenario The scenario to get the snapshot from.
22 * @param snapshot_id The id of the snapshot to get.

```

```

23  * @param snapshot A pointer to where a pointer to the snapshot
can be stored.
24  * @throws FAILURE if id not in snapshot list.
25  */
26 exception tekScenarioGetSnapshot(const TekScenario* scenario,
const uint snapshot_id, TekBodySnapshot** snapshot) {
27     // loop over the snapshots
28     // linear search...
29     const ListItem* item;
30     foreach(item, (&scenario->snapshots), {
31         const struct TekScenarioPair* pair = item->data;
32         if (pair->id == snapshot_id) {
33             *snapshot = pair->snapshot;
34             return SUCCESS;
35         }
36     });
37
38     // should only be reached if not found
39     tekThrow(FAILURE, "ID not in snapshot list");
40 }
41
42 /**
43  * Get a snapshot id/data by using the index at which its name
appears in the name list. Very useful in TekGuiList callbacks when
the user clicks the name of an object they want and you want the
associated snapshot -_,-
44  * @param scenario The scenario to get the snapshot from.
45  * @param name_index The index of the name in the name list.
46  * @param snapshot The snapshot to get, provide NULL if
unwanted.
47  * @param snapshot_id The id to get, provide NULL if unwanted
48  */
49 void tekScenarioGetByNameIndex(const TekScenario* scenario,
const uint name_index, TekBodySnapshot** snapshot, int* snapshot_id)
{
50     // if name index out of range, return invalid
51     if (name_index >= scenario->names.length) {
52         if (snapshot) *snapshot = NULL;
53         if (snapshot_id) *snapshot_id = -1;
54     }
55
56     // iterate over the list of names until reaching name index

```

```

57      // this is so we can get the ListItem pointer, will match
what the scenario pair points to
58      // cannot access directly due to being a linked list
59      const ListItem* item;
60      uint index = 0;
61      foreach(item, (&scenario->names), {
62          if (index == name_index)
63              break;
64          index++;
65      });
66
67      // find the scenario pair that has a matching list item
pointer as what we just found
68      const ListItem* search_item;
69      foreach(search_item, (&scenario->snapshots), {
70          const struct TekScenarioPair* pair = search_item->data;
71          if (pair->list_ptr == item) {
72              if (snapshot) *snapshot = pair->snapshot;
73              if (snapshot_id) *snapshot_id = (int)pair->id;
74              return;
75          }
76      });
77
78      // if no match, mark as being invalid.
79      if (snapshot) *snapshot = NULL;
80      if (snapshot_id) *snapshot_id = -1;
81  }
82
83 /**
84  * Get the name of a snapshot using its id.
85  * @param scenario The scenario from which to get the name
86  * @param snapshot_id The id of the snapshot for which to get
the name
87  * @param snapshot_name A pointer to where a pointer to the
buffer can be put
88  * @note The name is a direct pointer into the name list, so
don't edit unless you want changes to be reflected visually.
89  * @throws FAILURE if id not in snapshot list.
90  */
91 exception tekScenarioGetName(const TekScenario* scenario, const
uint snapshot_id, char** snapshot_name) {
92     // gotta loop over the list of items.
93     const ListItem* item;

```

```

94     foreach(item, (&scenario->snapshots), {
95         const struct TekScenarioPair* pair = item->data;
96         if (pair->id == snapshot_id) {
97             // found the name, so return pointer
98             *snapshot_name = pair->list_ptr->data;
99             return SUCCESS;
100        }
101    });
102    // if made it to the end of the list without returning,
then not found
103    tekThrow(FAILURE, "ID not in snapshot list");
104 }
105
106 /**
107  * Set the name of a snapshot in a scenario by id. Will copy
the name into a new buffer.
108 * @param scenario The scenario for which to set a name for.
109 * @param snapshot_id The id of the snapshot for which to
update the name
110 * @param snapshot_name The new name of the snapshot.
111 * @throws MEMORY_EXCEPTION if malloc() explodes
112 * @throws FAILURE if id is not in snapshot list
113 */
114 exception tekScenarioSetName(const TekScenario* scenario, const
uint snapshot_id, const char* snapshot_name) {
115     // need to iterate over each snapshot to find the one
associated with the specified id
116     const ListItem* item;
117     foreach(item, (&scenario->snapshots), {
118         const struct TekScenarioPair* pair = item->data;
119         if (pair->id == snapshot_id) {
120             // got it
121             // create a buffer that's big enough to fit the new
name
122             const uint len_name = strlen(snapshot_name) + 1;
123             char* new_name = realloc(pair->list_ptr->data,
len_name * sizeof(char));
124             if (!new_name)
125                 tekThrow(MEMORY_EXCEPTION, "Failed to
realloc memory for name.");
126
127             // copy in and update the name list
128             memcpy(new_name, snapshot_name, len_name);

```

```

129         pair->list_ptr->data = new_name;
130
131         return SUCCESS;
132     }
133 }
134 tekThrow(FAILURE, "ID not in snapshot list");
135 }
136
137 /**
138 * Get the next available ID in the scenario. Will first check
139 if any ids have been made available by removing old items, else will
140 return the lowest fresh id.
141 * @param scenario The scenario for which to get the next
142 available id.
143 * @param next_id A pointer to an unsigned integer which will
144 have the id written to it.
145 * @throws QUEUE_EXCEPTION if something horrible happens :P
146 */
147 exception tekScenarioGetNextId(TekScenario* scenario, uint*
next_id) {
148     // check if there is any unused ids. if not, we can just
149     // boom out the length of the snapshots
150     if (queueIsEmpty(&scenario->unused_ids)) {
151         *next_id = scenario->snapshots.length;
152         return SUCCESS;
153     }
154
155     // if non empty, then we dequeue the unused id queue to get
156     // an id to use.
157     void* next_id_ptr;
158     tekChainThrow(queueDequeue(&scenario->unused_ids,
159     &next_id_ptr));
160     *next_id = (uint)next_id_ptr;
161     return SUCCESS;
162 }
163
164 /**
165 * Create a scenario pair, this includes the data needed for
166 each snapshot such as the name, id and properties.
167 * @param scenario The scenario that the scenario pair is
168 associated with, where the name should be stored.
169 * @param copy_snapshot A body snapshot containing data to
170 copy.

```

```

161     * @param snapshot_id The id of the snapshot being created
162     * @param snapshot_name The name of the snapshot being created.
163     * @param pair A pointer to the pair which will be created.
164     * @throws MEMORY_EXCEPTION if malloc() fails.
165     */
166 static exception tekScenarioCreatePair(TekScenario* scenario,
167 const TekBodySnapshot* copy_snapshot, const uint snapshot_id, const
168 char* snapshot_name, struct TekScenarioPair** pair) {
169     // attempt to allocate memory for the new pair
170     *pair = malloc(sizeof(struct TekScenarioPair));
171     if (!*pair)
172         tekThrow(MEMORY_EXCEPTION, "Failed to allocate snapshot
173 pair.");
174     // now attempt to allocate memory for the body
175     (*pair)->snapshot =
176 (TekBodySnapshot*)malloc(sizeof(TekBodySnapshot));
177     if (!(*pair)->snapshot) {
178         free(*pair);
179         tekThrow(MEMORY_EXCEPTION, "Failed to allocate
180 snapshot.");
181     }
182     // now attempt to allocate memory for the name
183     const uint len_name = strlen(snapshot_name) + 1;
184     char* name = malloc(len_name * sizeof(char));
185     if (!name) {
186         free(*pair);
187         free((*pair)->snapshot);
188         tekThrow(MEMORY_EXCEPTION, "Failed to allocate snapshot
189 name.");
190     }
191     memcpy(name, snapshot_name, len_name);
192     // once everything is allocated, attempt to add the pair to
193     // the list of snapshots
194     tekChainThrowThen(listInsertItem(&scenario->names,
195 scenario->names.length - 1, name), {
196         free(*pair);
197         free((*pair)->snapshot);
198         free(name);
199     });

```

```

195     // after this, no more exceptions can occur, so we can
196     // start filling in the data
197     // set the list_ptr of the pair as the newly added name
198     ListItem* list_ptr = scenario->names.data;
199     while (list_ptr->next) {
200         if (!list_ptr->next->next) break;
201         list_ptr = list_ptr->next;
202     }
203     (*pair)->list_ptr = list_ptr;
204
205     // copy the snapshot data into the new buffer
206     memcpy((*pair)->snapshot, copy_snapshot,
207            sizeof(TekBodySnapshot));
208     (*pair)->id = snapshot_id;
209
210     return SUCCESS;
211 }
212 /**
213 * Delete a scenario pair, free allocated memory and the pair.
214 * @param pair The pair to delete.
215 */
216 static void tekScenarioDeletePair(struct TekScenarioPair* pair)
{
217     // free the snapshot data first, or else we lose it.
218     free(pair->snapshot);
219     free(pair);
220 }
221 /**
222 * Add a new snapshot or update an existing snapshot with a
223 * specified id. Will update the name and data associated with the id.
224 * @param scenario The scenario that should be added to
225 * @param copy_snapshot The body snapshot to copy into the
226 * scenario
227 * @param snapshot_id The id that should be associated with the
228 * snapshot
229 * @param snapshot_name The name of the snapshot being added.
230 * @throws MEMORY_EXCEPTION if malloc() fails.
231 */

```

```

230 exception tekScenarioPutSnapshot(TekScenario* scenario, const
TekBodySnapshot* copy_snapshot, const uint snapshot_id, const char*
snapshot_name) {
231     // first, check if a snapshot with the id already exists.
232     // need to check each id
233     const ListItem* item;
234     foreach(item, (&scenario->snapshots), {
235         struct TekScenarioPair* pair = item->data;
236         if (pair->id == snapshot_id) {
237             // if we find one with an existing id, need to
update it
238             // first, copy the data we want
239             memcpy(pair->snapshot, copy_snapshot, sizeof(struct
TekBodySnapshot));
240
241             // now, copy the new name into the name list
242             // need to reallocate cuz length might be
different.
243             const uint len_name = strlen(snapshot_name) + 1;
244             char* new_name = realloc(pair->list_ptr->data,
len_name * sizeof(char));
245             if (!new_name)
246                 tekThrow(MEMORY_EXCEPTION, "Failed to
realloc memory for name.");
247
248             // copy new name into list
249             memcpy(new_name, snapshot_name, len_name);
250             pair->list_ptr->data = new_name;
251
252             return SUCCESS;
253         }
254     });
255
256     // if the id doesn't exist, create a new snapshot with the
data.
257     struct TekScenarioPair* new_pair;
258     tekChainThrow(tekScenarioCreatePair(scenario,
copy_snapshot, snapshot_id, snapshot_name, &new_pair));
259     tekChainThrowThen(listAddItem(&scenario->snapshots,
new_pair), {
260         tekScenarioDeletePair(new_pair);
261     });
262

```

```

263     return SUCCESS;
264 }
265
266 /**
267  * Delete a snapshot, freeing any allocated memory of the
268  * struct.
269  * @param scenario The scenario to delete
270  * @param snapshot_id The id of the snapshot to delete
271  * @throws FAILURE if id does not exist.
272 */
273 exception tekScenarioDeleteSnapshot(TekScenario* scenario,
274 const uint snapshot_id) {
275     // first thing to do is remove scenario from list of
276     // scenarios.
277     // locate the scenario using its id.
278     const ListItem* item;
279     const ListItem* list_ptr = 0;
280     uint index = 0;
281     flag found = 0;
282     foreach(item, (&scenario->snapshots), {
283         struct TekScenarioPair* pair = item->data;
284         if (pair->id == snapshot_id) {
285             // if id matches, remove snapshot from list and
286             // mark this id as unused.
287             // allows us to fill gaps of ids when they're
288             // removed
289             tekChainThrow(listRemoveItem(&scenario->snapshots,
290             index, NULL));
291             tekChainThrow(queueEnqueue(&scenario->unused_ids,
292             (void*)snapshot_id));
293             list_ptr = pair->list_ptr;
294             free(pair);
295             found = 1;
296             break;
297         }
298     }
299     index++;
300 });
301
302 if (!found) tekThrow(FAILURE, "ID not in snapshot list");
303
304 // the other thing to do is remove it from the list of
305 names

```

```

298     // list_ptr should have been found in the last loop.
299     // list_ptr points to the name list where the name is
300     // stored.
301     index = 0;
302     found = 0;
303     foreach(item, (&scenario->names), {
304         if (item == list_ptr) {
305             found = 1;
306             break;
307         }
308         index++;
309     });
310
311     if (!found)
312         return SUCCESS;
313
314     // finally, remove the name
315     tekChainThrow(listRemoveItem(&scenario->names, index,
316     NULL));
316     return SUCCESS;
317 }
318
319 /**
320 * Create a new scenario, allocates some memory for different
321 internal structures.
322 * @param scenario The scenario to create.
323 * @throws MEMORY_EXCEPTION if malloc() fails
324 */
325 exception tekCreateScenario(TekScenario* scenario) {
326     // initialise some internal structures
327     listCreate(&scenario->snapshots);
328     listCreate(&scenario->names);
329     queueCreate(&scenario->unused_ids);
330
331     // add the button to add a new object
332     // shouldn't really be done here. ^\_(ツ)_/-
333     const char* new_object_string = "Add New Object";
334     const uint len_new_object_string =
335     strlen(new_object_string) + 1;
336     char* new_object_buffer = (char*)malloc(sizeof(char) *
337     len_new_object_string);

```

```

335     if (!new_object_buffer)
336         tekThrow(MEMORY_EXCEPTION, "Failed to allocate buffer
for new object string.");
337     memcpy(new_object_buffer, new_object_string,
len_new_object_string);
338     tekChainThrowThen(listAddItem(&scenario->names,
new_object_buffer), {
339         free(new_object_buffer);
340     });
341
342     return SUCCESS;
343 }
344
345 /**
346 * Scan a single snapshot from a buffer. Expects the same
format as specified by SNAPSHOT_READ_FORMAT - key value pairs of
snapshot data seperated by new lines.
347 * @param string The input buffer to scan from
348 * @param snapshot The snapshot to write into.
349 * @param snapshot_id An unsigned int to write the snapshot id
into.
350 * @param snapshot_name An allocated char buffer to write the
name of the snapshot into.
351 * @return Number of items successfully scanned.
352 */
353 static int tekScanSnapshot(const char* string, TekBodySnapshot*
snapshot, uint* snapshot_id, char* snapshot_name) {
354     // wrapper around sscanf
355     // just uses SNAPSHOT_READ_FORMAT for consistency.
356     // also helps cuz you dont have to access the elemets of
snapshot every time
357     return sscanf(
358         string,
359         SNAPSHOT_READ_FORMAT,
360         snapshot_id,
361         snapshot_name,
362         &snapshot->position[0], &snapshot->position[1],
&snapshot->position[2],
363         &snapshot->rotation[0], &snapshot->rotation[1],
&snapshot->rotation[2], &snapshot->rotation[3],
364         &snapshot->velocity[0], &snapshot->velocity[1],
&snapshot->velocity[2],
365         &snapshot->mass,

```

```

366         &snapshot->friction,
367         &snapshot->restitution,
368         &snapshot->immovable,
369         snapshot->model,
370         snapshot->material
371     );
372 }
373
374 /**
375  * Read a scenario from a specified file, and load it into a
376  * scenario struct.
377  * @param scenario_filepath The path of the file containing the
378  * scenario data.
379  * @param scenario A pointer to a scenario to fill with the
380  * data in the file.
381  * @throws MEMORY_EXCEPTION if malloc() fails.
382  */
383 exception tekReadScenario(const char* scenario_filepath,
TekScenario* scenario) {
384     // initialise the scenario structure, initialise some
internals.
385     tekChainThrow(tekCreateScenario(scenario));
386
387     // get the size of the file and attempt to write buffer
with the data
388     uint len_file;
389     tekChainThrow(getFileSize(scenario_filepath, &len_file));
390     char* file = (char*)malloc(len_file * sizeof(char));
391     if (!file)
392         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
to read file into.");
393     tekChainThrow(readFile(scenario_filepath, len_file, file));
394
395     // iterate over the file, looking for new lines.
396     // SNAPSHOT_NUM_LINES is the number of lines per snapshot,
so every "that many" lines we need to add a new object
397     // every new line signals that a new piece of data is added
to the current object
398     // this is mostly handled by the tekScanSnapshot / scanf
methods tho
399     char* c = file;
400     char* scenario_start = file;
401     uint line_number = 0;

```

```

399     while (*c) {
400         // count new lines
401         if (*c == '\n')
402             line_number++;
403
404         // once reaching magic number of newlines, scan the
last bit of file into a snapshot
405         if (*c == '\n' && line_number % SNAPSHOT_NUM_LINES ==
0) {
406             // build a new snapshot, need some space to write
stuff into
407             TekBodySnapshot snapshot = {};
408             uint snapshot_id = 0;
409             // absolute bodge, i have no idea how to find the
length of the name before scanning :)
410             // ples dont name anything using more than 256
chars
411             char snapshot_name[256];
412             snapshot.model = malloc(256 * sizeof(char));
413             snapshot.material = malloc(256 * sizeof(char));
414             // now scan and write snapshot
415             if (tekScanSnapshot(scenario_start, &snapshot,
&snapshot_id, snapshot_name) < 0) {
416                 free(file);
417                 tekThrow(FAILURE, "Failed to read snapshot
file.");
418             }
419             tekChainThrowThen(tekScenarioPutSnapshot(scenario,
&snapshot, snapshot_id, snapshot_name), {
420                 free(file);
421             });
422             // go to next char in buffer
423             scenario_start = c + 1;
424         }
425
426         c++;
427     }
428
429     return SUCCESS;
430 }
431
432 /**

```

```

433  * Write a single snapshot of a body to a buffer. Also returns
434  * the number of bytes written in the buffer.
435  * @param string A pointer to the buffer where the snapshot
436  * should be written.
437  * @param max_length The maximum number of bytes that can be
438  * written (length of buffer?)
439  * @param snapshot A pointer to the snapshot to be written.
440  * @param snapshot_id The id of the snapshot.
441  * @param snapshot_name The name of the snapshot
442  * @return The number of bytes written.
443  */
444 static int tekWriteSnapshot(char* string, size_t max_length,
445 const TekBodySnapshot* snapshot, const uint snapshot_id, const char*
446 snapshot_name) {
447     // wrapper around snprintf, just reduces chance of error
448     // when reusing this func.
449     // also, allows formatting to be changed more easily.
450     return snprintf(
451         string, max_length,
452         SNAPSHOT_WRITE_FORMAT,
453         snapshot_id,
454         snapshot_name,
455         EXPAND_VEC3(snapshot->position),
456         EXPAND_VEC4(snapshot->rotation),
457         EXPAND_VEC3(snapshot->velocity),
458         snapshot->mass,
459         snapshot->friction,
460         snapshot->restitution,
461         snapshot->immovable,
462         snapshot->model,
463         snapshot->material
464     );
465     //
466 }
467 /**
468  * Allocate a buffer containing the ids of all snapshots in the
469  * scenario.
470  * @param scenario The scenario to generate this buffer for.
471  * @param ids A pointer to where a pointer to the newly created
472  * buffer will be written.
473  * @param num_ids A pointer to where the number of ids in the
474  * new buffer will be written.

```

```

467     * @throws MEMORY_EXCEPTION if malloc() fails.
468     * @note This function allocates memory which you are
469     * responsible for freeing.
470 exception tekScenarioGetAllIds(const TekScenario* scenario,
471     uint** ids, uint* num_ids) {
472     // allocate a buffer big enough to fit all the ids of this
473     // scenario
474     // will equal the number of objects in the scenario, hence
475     // the use of length.
476     const uint num_snapshots = scenario->snapshots.length;
477     *num_ids = num_snapshots;
478     *ids = (uint*)malloc(num_snapshots * sizeof(uint));
479     if (!*ids)
480         tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for id list.");
481
482     // loop through "scenario pairs", which contain the id
483     const ListItem* item;
484     uint index = 0;
485     foreach(item, (&scenario->snapshots), {
486         // for each pair, get the id and add it to the array
487         const struct TekScenarioPair* pair = item->data;
488         (*ids)[index] = pair->id;
489         index++;
490     });
491
492 /**
493     * Write the data stored in a scenario to a specified filepath.
494     * @param scenario The scenario which should have its data
written.
495     * @param scenario_filepath The path of the file to write the
scenario to.
496     * @throws FILE_EXCEPTION if the file does not exist or cannot
be written.
497     * @throws MEMORY_EXCEPTION if malloc() fails.
498 */
499 exception tekWriteScenario(const TekScenario* scenario, const
char* scenario_filepath) {

```

```

500      // for each object in the scenario, calculate the number of
bytes needed to write it.
501      const ListItem* item;
502      int len_buffer = 0;
503      foreach(item, (&scenario->snapshots), {
504          // writing to NULL, max length 0 -> get the number of
characters needed
505          const struct TekScenarioPair* pair = (struct
TekScenarioPair*)item->data;
506          len_buffer += tekWriteSnapshot(NULL, 0, pair->snapshot,
pair->id, pair->list_ptr->data);
507      });
508
509      // null-terminated string, so need to add the last
character (=0)
510      len_buffer += 1;
511
512      // mallocate enough memory to write to
513      char* buffer = (char*)malloc(len_buffer * sizeof(char));
514      if (!buffer)
515          tekThrow(MEMORY_EXCEPTION, "Failed to allocate memory
for scenario file.");
516
517      // now loop back over, writing to the buffer.
518      // increment the write pointer after each item, so we dont
just overwrite the first item over and over.
519      int write_ptr = 0;
520      uint id = 0;
521      foreach(item, (&scenario->snapshots), {
522          const struct TekScenarioPair* pair = (struct
TekScenarioPair*)item->data;
523          printf("[[[\n%s\n]]]\n", (char*)pair->list_ptr->data);
524          write_ptr += tekWriteSnapshot(buffer + write_ptr,
len_buffer - write_ptr, pair->snapshot, id++, pair->list_ptr->data);
525      });
526
527      // finally, write this buffer to the file.
528      tekChainThrow(writeFile(buffer, scenario_filepath));
529
530      free(buffer);
531      return SUCCESS;
532 }
533

```

```

534 /**
535  * Delete a scenario, removing any allocated memory of the
536  * struct.
537  */
538 void tekDeleteScenario(TekScenario* scenario) {
539     // iterate over objects in scenario and free them.
540     ListItem* item;
541     foreach(item, (&scenario->snapshots), {
542         struct TekScenarioPair* pair = item->data;
543         tekScenarioDeletePair(pair);
544     });
545
546     // now free the list that stored the objects
547     // and free helper structures
548     listDelete(&scenario->snapshots);
549     listFreeAllData(&scenario->names);
550     listDelete(&scenario->names);
551     queueDelete(&scenario->unused_ids);
552 }

```

tekphys/scenario.h

```

1 #pragma once
2
3 #include "body.h"
4 #include "../tekg1.h"
5 #include "../core/exception.h"
6 #include "../core/list.h"
7 #include "../core/queue.h"
8
9 typedef struct TekScenario {
10     List snapshots;
11     List names;
12     Queue unused_ids;
13 } TekScenario;
14
15 exception tekCreateScenario(TekScenario* scenario);
16 exception tekReadScenario(const char* scenario_filename,
TekScenario* scenario);
17 exception tekWriteScenario(const TekScenario* scenario, const
char* scenario_filename);
18

```

```

19 exception tekScenarioGetSnapshot(const TekScenario* scenario,
uint snapshot_id, TekBodySnapshot** snapshot);
20 void tekScenarioGetByNameIndex(const TekScenario* scenario, uint
name_index, TekBodySnapshot** snapshot, int* snapshot_id);
21
22 exception tekScenarioGetName(const TekScenario* scenario, uint
snapshot_id, char** snapshot_name);
23 exception tekScenarioSetName(const TekScenario* scenario, uint
snapshot_id, const char* snapshot_name);
24
25 exception tekScenarioPutSnapshot(TekScenario* scenario, const
TekBodySnapshot* copy_snapshot, uint snapshot_id, const char*
snapshot_name);
26 exception tekScenarioDeleteSnapshot(TekScenario* scenario, uint
snapshot_id);
27 exception tekScenarioGetNextId(TekScenario* scenario, uint*
next_id);
28 exception tekScenarioGetAllIds(const TekScenario* scenario,
uint** ids, uint* num_ids);
29
30 void tekDeleteScenario(TekScenario* scenario);

```

shader/button.glvs

```

1 #version 330 core
2
3 layout (location = 0) in vec2 minmax_x;
4 layout (location = 1) in vec2 minmax_y;
5 layout (location = 2) in vec2 minmax_ix;
6 layout (location = 3) in vec2 minmax_iy;
7
8 uniform vec2 window_size;
9 uniform vec4 bg_colour;
10 uniform vec4 bd_colour;
11
12 out VS_OUT {
13     vec2 minmax_x;
14     vec2 minmax_y;
15     vec2 minmax_ix;
16     vec2 minmax_iy;
17     vec4 bg_colour;
18     vec4 bd_colour;
19 } vs_out;
20

```

```

21 float remap(float value, float max_value) {
22     return value / max_value * 2.0f - 1.0f;
23 }
24
25 void main() {
26     gl_Position = vec4(minmax_x.x, minmax_y.x, 0.0f, 1.0f);
27     vs_out.minmax_x = vec2(remap(minmax_x.x, window_size.x),
28     remap(minmax_x.y, window_size.x));
29     vs_out.minmax_y = vec2(-remap(minmax_y.y, window_size.y), -
30     remap(minmax_y.x, window_size.y));
31     vs_out.minmax_ix = vec2(remap(minmax_ix.x, window_size.x),
32     remap(minmax_ix.y, window_size.x));
33     vs_out.minmax_iy = vec2(-remap(minmax_iy.y, window_size.y), -
34     -remap(minmax_iy.x, window_size.y));
35 }
```

shader/fragment.glfs

```

1 #version 330 core
2
3 in vec3 f_position;
4 in vec3 f_normal;
5
6 out vec4 color;
7
8 uniform vec3 light_color;
9 uniform vec3 camera_pos;
10 uniform vec3 light_position;
11
12 void main() {
13     vec3 ambient = light_color * 0.2f;
14
15     vec3 light_direction = normalize(light_position -
16     f_position);
16     vec3 diffuse = light_color * max(dot(f_normal,
17     light_direction), 0.0f);
18
18     vec3 view_direction = normalize(camera_pos - f_position);
19     vec3 reflect_direction = reflect(-light_direction,
19     f_normal);
```

```
20     float spec_modifier = pow(max(dot(view_direction,
reflect_direction), 0.0), 32);
21     vec3 specular = 0.5 * spec_modifier * light_color;
22
23     color = vec4(ambient + diffuse + specular, 1.0f);
24 }
```

shader/image_vertex.glvs

```
1 #version 330 core
2
3 layout (location = 0) in vec2 position;
4 layout (location = 1) in vec2 v_texture_uv;
5
6 uniform mat4 projection;
7 uniform vec2 start_position;
8
9 out vec2 f_texture_uv;
10
11 void main() {
12     gl_Position = projection * vec4(position.x +
start_position.x, position.y + start_position.y, 0.0f, 1.0f);
13     f_texture_uv = v_texture_uv;
14 }
```

shader/image_fragment.glfs

```
1 #version 330 core
2
3 in vec2 f_texture_uv;
4
5 uniform sampler2D texture_sampler;
6
7 out vec4 color;
8
9 void main() {
10     color = texture(texture_sampler, f_texture_uv);
11 }
```

shader/line_vertex.glvs

```
1 #version 330 core
2
3 layout (location = 0) in vec2 position;
4
5 uniform mat4 projection;
```

```
6
7 void main() {
8     gl_Position = projection * vec4(position.x, position.y, 0.0f,
1.0f);
9 }
```

shader/line_fragment.glfs

```
1 #version 330 core
2
3 out vec4 color;
4
5 uniform vec4 line_color;
6
7 void main() {
8     color = line_color;
9 }
```

shader/oval_vertex.glvs

```
1 #version 330 core
2
3 layout (location = 0) in vec2 position;
4
5 out vec2 frag_pos;
6
7 uniform mat4 projection;
8
9 void main() {
10     gl_Position = projection * vec4(position.x, position.y,
0.0f, 1.0f);
11     frag_pos = position;
12 }
```

shader/oval_fragment.glfs

```
1 #version 330 core
2
3 in vec2 frag_pos;
4
5 out vec4 color;
6
7 uniform float inv_width;
8 uniform float inv_height;
9 uniform float min_dist;
10 uniform vec4 oval_color;
```

```

11 uniform vec2 center;
12
13 void main() {
14     vec2 delta = frag_pos - center;
15     delta.x *= inv_width;
16     delta.y *= inv_height;
17     float distance = delta.x * delta.x + delta.y * delta.y;
18     if (distance < 1.0f && distance > min_dist) {
19         color = oval_color;
20     } else {
21         color = vec4(0.0f, 0.0f, 0.0f, 0.0f);
22     }
23 }
```

shader/texture.glvs

```

1 #version 330 core
2
3 layout (location = 0) in vec3 v_position;
4 layout (location = 1) in vec3 v_normal;
5 layout (location = 2) in vec2 v_tex;
6
7 out vec3 f_position;
8 out vec3 f_normal;
9 out vec2 f_tex;
10
11 uniform mat4 model;
12 uniform mat4 view;
13 uniform mat4 projection;
14
15 void main() {
16     f_position = vec3(model * vec4(v_position, 1.0f));
17     f_normal = mat3(transpose(inverse(model))) * v_normal;
18     gl_Position = projection * view * vec4(f_position, 1.0f);
19     f_tex = v_tex;
20 }
```

shader/texture.glfss

```

1 #version 330 core
2
3 in vec3 f_position;
4 in vec3 f_normal;
5 in vec2 f_tex;
6
```

```

7  out vec4 color;
8
9  uniform vec3 light_color;
10 uniform vec3 camera_pos;
11 uniform vec3 light_position;
12 uniform sampler2D texture_file;
13
14 void main() {
15     vec3 tex_colour = vec3(texture(texture_file, f_tex));
16     vec3 ambient = tex_colour * light_color * 0.2f;
17
18     vec3 light_direction = normalize(light_position -
f_position);
19     vec3 diffuse = tex_colour * light_color * max(dot(f_normal,
light_direction), 0.0f);
20
21     vec3 view_direction = normalize(camera_pos - f_position);
22     vec3 reflect_direction = reflect(-light_direction,
f_normal);
23     float spec_modifier = pow(max(dot(view_direction,
reflect_direction), 0.0), 32);
24     vec3 specular = 0.5 * spec_modifier * light_color;
25
26     color = vec4(ambient + diffuse + specular, 1.0f);
27 }

```

shader/text_vertex.glvs

```

1 #version 330 core
2
3 layout (location = 0) in vec2 position;
4 layout (location = 1) in vec2 tex_coord_v;
5
6 out vec2 tex_coord_f;
7
8 uniform mat4 projection;
9 uniform float draw_x;
10 uniform float draw_y;
11
12 void main() {
13     gl_Position = projection * vec4(draw_x + position.x, draw_y
+ position.y, 0.0f, 1.0f);
14     tex_coord_f = tex_coord_v;
15 }

```

shader/text_fragment.glfs

```
1 #version 330 core
2
3 in vec2 tex_coord_f;
4
5 out vec4 colour;
6
7 uniform sampler2D atlas;
8 uniform vec4 text_colour;
9
10 void main() {
11     vec4 sampled_colour = texture(atlas, tex_coord_f);
12     colour = vec4(text_colour.r, text_colour.g, text_colour.b,
min(text_colour.a, sampled_colour.r));
13 }
```

shader/vertex.glvs

```
1 #version 330 core
2
3 layout (location = 0) in vec3 v_position;
4 layout (location = 1) in vec3 v_normal;
5
6 out vec3 f_position;
7 out vec3 f_normal;
8
9 uniform mat4 model;
10 uniform mat4 view;
11 uniform mat4 projection;
12
13 void main() {
14     f_position = vec3(model * vec4(v_position, 1.0f));
15     f_normal = mat3(transpose(inverse(model))) * v_normal;
16     gl_Position = projection * view * vec4(f_position, 1.0f);
17 }
```

shader/window.glvs

```
1 #version 330 core
2
3 layout (location = 0) in vec2 minmax_x;
4 layout (location = 1) in vec2 minmax_y;
5 layout (location = 2) in vec2 minmax_ix;
6 layout (location = 3) in vec2 minmax_iy;
7 layout (location = 4) in vec4 bg_colour;
```

```

8  layout (location = 5) in vec4 bd_colour;
9
10 uniform vec2 window_size;
11
12 out VS_OUT {
13     vec2 minmax_x;
14     vec2 minmax_y;
15     vec2 minmax_ix;
16     vec2 minmax_iy;
17     vec4 bg_colour;
18     vec4 bd_colour;
19 } vs_out;
20
21 float remap(float value, float max_value) {
22     return value / max_value * 2.0f - 1.0f;
23 }
24
25 void main() {
26     gl_Position = vec4(minmax_x.x, minmax_y.x, 0.0f, 1.0f);
27     vs_out.minmax_x = vec2(remap(minmax_x.x, window_size.x),
28     remap(minmax_x.y, window_size.x));
29     vs_out.minmax_y = vec2(-remap(minmax_y.y, window_size.y), -
30     remap(minmax_y.x, window_size.y));
31     vs_out.minmax_ix = vec2(remap(minmax_ix.x, window_size.x),
32     remap(minmax_ix.y, window_size.x));
33     vs_out.minmax_iy = vec2(-remap(minmax_iy.y, window_size.y),
34     -remap(minmax_iy.x, window_size.y));
35 }

```

shader/window.glsl

```

1 #version 330 core
2
3 layout (points) in;
4 layout (triangle_strip, max_vertices = 6) out;
5
6 in VS_OUT {
7     vec2 minmax_x;
8     vec2 minmax_y;
9     vec2 minmax_ix;

```

```

10     vec2 minmax_iy;
11     vec4 bg_colour;
12     vec4 bd_colour;
13 } vs_out[];
14
15 out vec4 bg_colour;
16 out vec4 bd_colour;
17
18 out vec2 minmax_ix;
19 out vec2 minmax_iy;
20 out vec2 frag_pos;
21
22 void emitTriangle(vec2 vertices[4], int a, int b, int c) {
23     gl_Position = vec4(vertices[a], 0.0f, 1.0f);
24     frag_pos = vertices[a];
25     EmitVertex();
26     gl_Position = vec4(vertices[b], 0.0f, 1.0f);
27     frag_pos = vertices[b];
28     EmitVertex();
29     gl_Position = vec4(vertices[c], 0.0f, 1.0f);
30     frag_pos = vertices[c];
31     EmitVertex();
32     EndPrimitive();
33 }
34
35 void main() {
36     vec2 vertices[4];
37
38     for (int i = 0; i < 2; i++) {
39         for (int j = 0; j < 2; j++) {
40             if (i == 0) {
41                 vertices[i + 2 * j].x = vs_out[0].minmax_x.x;
42             } else {
43                 vertices[i + 2 * j].x = vs_out[0].minmax_x.y;
44             }
45             if (j == 0) {
46                 vertices[i + 2 * j].y = vs_out[0].minmax_y.x;
47             } else {
48                 vertices[i + 2 * j].y = vs_out[0].minmax_y.y;
49             }
50         }
51     }
52 }
```

```

53     bg_colour = vs_out[0].bg_colour;
54     bd_colour = vs_out[0].bd_colour;
55
56     minmax_ix = vs_out[0].minmax_ix;
57     minmax_iy = vs_out[0].minmax_iy;
58
59     emitTriangle(vertices, 0, 1, 2);
60     emitTriangle(vertices, 1, 3, 2);
61 }

```

shader/window.glfs

```

1 #version 330 core
2
3 in vec4 bg_colour;
4 in vec4 bd_colour;
5
6 in vec2 minmax_ix;
7 in vec2 minmax_iy;
8 in vec2 frag_pos;
9
10 out vec4 colour;
11
12 void main() {
13     if (frag_pos.x > minmax_ix.x && frag_pos.x < minmax_ix.y &&
frag_pos.y > minmax_iy.x && frag_pos.y < minmax_iy.y) {
14         colour = bg_colour;
15     } else {
16         colour = bd_colour;
17     }
18 }

```

Testing

Test Strategy

To ensure the robustness of my system, I will begin by testing the base modules, especially the data structures I have implemented (for example "hashtable.c", "vector.c", "list.c" and other files in "/core"). This will allow me to be confident that they work elsewhere in the program as I continue with the rest of my testing. In order to do this, I will be using a suite of unit tests to cover some valid and invalid operations on the modules to see if they function correctly under all circumstances.

After this, I will move on to testing the GUI system. I will do this by simulating the actions of a potential user who is acting in the intended way, and then by acting in an unintended way. This could include moving GUI elements to places that they shouldn't be, entering non numeric values into numeric inputs, and otherwise attempting to break my UI.

Once this test is complete, I will move on to testing the physics simulation. I will do this by creating some basic scenarios on paper, and using the program to recreate these scenarios in the game and seeing if the output produces a similar result as to what is predicted on paper. This will be done by comparing the expected initial/final speeds.

Test Plan – Unit Test

Test Name	Test Purpose	Test Data	Expected	Actual
U1	Make sure that adding an item to a vector increases its length by one.	The number 12345.	New length is 1.	New length was 1.
U2	Make sure that the correct item is added to the vector.	The number 12345	The first item is 12345.	The first item was 12345.
U3	Make sure that the internal size was 1, which is what we requested when making the vector.	A new vector.	The internal size is 1.	The internal size was 1.
U4	Make sure that the size of the	The integers 0-9.	The size of the vector internally	The size of the vector internally

	vector increases when adding more items than it can currently store.		is greater than or equal to 10.	was greater than or equal to 10.
U5	Make sure that getting an integer from a list gets the right integer.	The numbers 4543, 5832, 66, 1453, 8936, 4936, 1709, 5623, 7166 and 3425.	The first item is 4543.	The first item was 4543.
U6	//	//	The sixth item is 4936.	The sixth item was 4936.
U7	Make sure that getting a pointer to an integer in a vector gets the right integer.	//	The pointer to the first item is 4543	The first item pointed to 4543.
U8	//	//	The pointer to the sixth item is 4936.	The sixth item pointed to 4936.
U9	Check that setting an integer in a vector changes the value.	The number 999.	The number at index 2 is now 999.	The number at index 2 was 999.
U10	Make sure that inserting an integer causes the other numbers after it to move.	The number 999.	The number at index 0 is now 999.	The number at index 0 was 999.
U11	//	//	The number at index 1 is now what was at index 0.	The number at index 1 was what was at index 0.
U12	//	//	The number at index 2 is now what was at index 1.	The number at index 2 was what was at index 1.
U13	Make sure that	A vector	Removing index	Removing index

	removing an item from a vector returns the value of what was there before.	containing 10, 20 and 30.	1 returns 20.	1 returned 20.
U14	Make sure that removing an item from a vector reduces its length by 1.	//	The length is now 2.	The length was 2.
U15	Make sure that removing an item from a vector doesn't affect the other items.	//	The first item is still 10.	The first item was still 10.
U16	Make sure that items after the removed one are moved down to fill the gap.	//	The second item is now 30.	The second item was 30.
U17	Make sure that an item can be popped successfully from a vector.	A vector containing 11, 22 and 33.	Popping an item returns true.	Popping an item returned true.
U18	Make sure that popping the vector returns the last item.	//	The popped item is 33.	The popped item was 33.
U19	Make sure that popping the vector reduces the length by one.	//	The new length is 2.	The new length was 2.
U20	Make sure that clearing a vector makes the length change to 0.	A vector containing 123, 123, 123.	The new length is 0.	The new length was 0.
U21	Make sure that	A vector	The last item is	The last item

	getting the last item in a vector works.	containing 1 and 2.	2. was 2.	
U22	Make sure that getting an item out of bounds returns an exception.	//	A vector exception is thrown.	A vector exception was thrown.
U23	Make sure that data of a different size to an integer can be gotten.	A vector containing structs of int, char* and void*.	The first item matches what was added first.	The first item matched what was added first.
U24	//	//	The second item matches what was added second.	The second item matched what was added second.
U25	Make sure that pointers to data of different size to an integer can be gotten.	//	The data pointed to by the third item matches what was added third.	The data pointed to by the third item matched what was added third.
U26	Make sure that creating a vector with a data size of 0 is invalid.	Size of 0.	A vector exception is thrown.	A vector exception is thrown.
U27	Make sure that a list can have items added to and gotten from.	A list which has the numbers 10, 20 and 30 added to it.	When retrieving the items, they match what was added.	The items match what was added.
U28	Items can be inserted into a list and items in a list can be updated.	A list containing the numbers 1 and 3.	Inserting the number 2 at index 1 makes the list 1, 2, 3, and the index 1 can be changed to the number 99.	The list was updated to 1, 2, 3 and the number 2 was changed to 99.
U29	Items can be removed by index from a list and can also be	A list containing the numbers 5, 6 and 7, the index 1 is removed to	Removing index 6 removes the 6, popping after this removes the 7.	The 6 was removed, followed by the 7 when removing

	popped from the end of the list.	retrieve the 6, and the list is popped to get the 7.	7	index 1 and popping afterwards.
U30	Moving an item in a list from one index to another works.	The list 1, 2, 3, where index 0 is moved to index 2.	Produces the list 2, 3, 1.	The list 2, 3, 1 was produced.
U31	Freeing all data from a list will not affect the length, as it does not delete the items only frees them.	A list of two dynamically allocated pointers.	The list remains at a length of two after freeing each item.	The list remained at length 2.
U32	Boundary and invalid tests, attempt to get an index out of range.	A list containing the number 42.	Getting the index 0 will give 42, getting index 5 will throw an error.	Index 0 gave 42, getting index 5 threw an error.
U33	Attempt to push and pop some items to a stack.	A stack with items 10, 20 and 30 pushed in that order.	The length is originally 3, popping the stack thrice will give 30, 20 and 10, and the length is reduced to 0.	The length was 3, the items 30, 20 and 10 were popped in that order, and the length was reduced to 0.
U34	Attempt to peek a stack.	A stack with the item 99 pushed to it.	The item 99 is returned, and the length is still 1.	The item 99 was returned, and the length was still 1.
U35	Boundary and invalid tests, attempt to peek and pop an empty stack.	An empty stack.	Attempting to peek or pop the stack will throw an exception.	Attempting to peek or pop the stack threw an exception.
U36	A queue can have items enqueued or dequeued.	A queue that has had the items 10, 20 and 30 enqueued in that order.	Dequeueing the queue will give the items 10, 20 and 30 in that order, and the	The items 10, 20 and 30 were dequeued in that order. The queue became

			queue will become empty.	empty.
U37	Peeking the queue will give the next item to dequeue without removing it.	A queue that has had the number 99 enqueued.	Peeking the queue will give the number 99 without changing the length.	Peeking the queue gave the number 99 and the length stayed at 1.
U38	If the queue has no items, it should say that it is empty.	An initially empty/ queue.	Should say it is empty, until an item (42) is added, should say not empty, after dequeuing was empty again.	The queue initially stated it was empty, until something was queued, then should be empty after again.
U39	Invalid and boundary tests for a queue, attempt to peek and dequeue an empty queue.	An empty queue.	Peeking or dequeuing the queue should throw an exception.	Peeking or dequeuing the queue threw an exception.
U40	A priority queue should be able to queue and dequeue items.	A priority queue with the items 10, 20 and 30 queued with respective priorities 3.0, 1.0 and 2.0.	The item with lowest priority value should be dequeued first, followed by the second least, etc.	The items 20, 30 and 10 were dequeued in that order, which have priorities 1.0, 2.0 and 3.0 as expected.
U41	Peeking a priority queue should reveal the item in the queue with the least priority value.	A queue with the items 5, 999 with priority 1.0 and 5.0 respectively.	Peeking should give the item 5, and the length should remain as 2.	5 was peeked, and the length stayed at 2.
U41	Boundary and invalid tests on priority queues, attempting to peek and dequeue an empty queue.	An empty priority queue.	Peeking or dequeuing from an empty priority queue will throw an exception.	Peeking or dequeuing an empty priority queue threw an exception.

U42	Check if a priority queue knows if it is empty.	An empty priority queue.	Should return that the priority queue is empty, until an item is queued, and should say is empty once again if the item is dequeued.	The priority queue said it was empty until an item was enqueued, and said it was empty once the item was dequeued.
U43	Ensure that a bit set can have individual bits set and checked.	A bit set with all bits set to 0.	The bits 0, 63, 64 and 127 are set, and those same bits as well as 14 and 120 are checked, only 14 and 120 are unset.	The bits 0, 63, 64 and 127 were set, the bits 14 and 120 were unset.
U44	Boundary and invalid tests for a bit set, attempt to set a bit outside of the range.	A bit set that has had its size fixed at 64.	Setting bit 63 is fine, setting bit 64 throws an exception.	Setting bit 63 was fine, setting bit 64 threw an exception.
U45	Clearing a bit set will return all bits to being unset.	A bit set which has had the bits 3, 17 and 31 set.	Clearing the bit set will make all bits of the bit set be unset.	All bits of the bit set were unset following the clearing.
U46	Setting the grows flag of the bit set will allow it to dynamically allocate more memory if a bit is set outside the range.	A bit set that does not grow and a bit set that does grow, both of initial size 32.	Setting the index 10 will work for both, setting the index 999 will throw an exception for the non-growing bit set but not for the growing one.	Setting the index 10 worked for both, setting index 999 threw an exception for the non-growing bit set.
U47	The bit set should also operate when using two-dimensional indexing.	A bit set with the indices (0, 0), (3, 2) and (15, 15) set.	Checking the indices (0, 0), (3, 2) and (15, 15) will all be set.	The indices (0, 0), (3, 2) and (15, 15) were all set.

U48	A hash table should be able to have items set and retrieved.	A hash table with the keys "alpha", "beta" and "charlie", with the values 100, 200 and 300.	Getting the key "alpha" will give 100, "beta" will give 200 and "charlie" will give 300.	"alpha" gave 100, "beta" gave 200, and "charlie" gave 300.
U49	A hash table should have the ability to check if a key exists and be able to remove it.	A hash table with the keys "foo" and "bar", with the values 111 and 222.	Should say keys "foo" and "bar" exist, while "baz" does not. Removing "foo" should update so that "foo" does no longer exist.	Keys "foo" and "bar" were said to exist, while "baz" was not. After removing "foo", only "bar" was said to exist.
U50	A hash table should have the ability to return an array containing all keys and an array with all values.	A hash table with the keys "one", "two" and "three", with associated values 10, 20 and 30.	Getting the keys should return an array containing "one", "two" and "three" in any order. Getting the values should return an array of 10, 20 and 30 in an order corresponding to the order of the keys.	The array of keys contained "one", "two" and "three", and the values 10, 20 and 30 were in the same order as their corresponding keys appeared in the keys array.
U51	Boundary and invalid tests for hash tables, attempt to get a missing key, set with a blank key and set null data.	An empty hash table.	Attempting to get the key "missing" should throw an exception, while adding with a key of "", and a value of NULL should work.	Attempting to get the key "missing" threw an exception, while setting the key "", or the value NULL, worked without exception.
U52	A hash table should be able to overwrite an existing key.	A hash table containing the key "testkey" with the value 42.	Setting "testkey" to be 999 should mean that subsequently getting "testkey"	After setting "testkey" to 999, getting "testkey" returned 999

			will now return 999.	instead of 42.
U53	A thread queue should be able to enqueue and dequeue items.	A thread queue with the items 10, 20, 30, 40 and 50 enqueued.	Dequeuing the thread queue will return 10, 20, 30, 40 and 50 in that order.	Dequeuing returned 10, 20, 30, 40 and 50 in that order.
U54	A thread queue should be able to transfer data between two threads.	Two threads with access to a shared thread queue.	Enqueueing a stream of data from one thread will lead to the same stream being received in the other thread.	Enqueueing a stream of data in one queue led to the same stream of data being received in the other thread.
U55	A thread queue should operate under abnormal circumstances, such as dequeuing an empty queue.	An empty thread queue.	Peeking or dequeuing the thread queue will throw an exception.	An exception was thrown when the empty thread queue was peeked or dequeued.
U56	Thread queues should function under heavy stress.	An empty thread queue.	The queue should function when enqueueing and dequeuing at maximum rate.	The queue functioned flawlessly under heavy load.
U57	Getting the size of a known file.	A file with known size of 10 bytes.	Getting the size of the file will give 11, which is the size of a string buffer needed to load this file.	The returned size was 11, which is 10 bytes of file plus a byte of null terminator.
U58	Reading a file should give the exact contents of the file.	A file with the lyrics to the song "Take me away".	The text read from the file matches exactly with the lyrics.	The text read matched the lyrics exactly.
U59	Reading an empty file will still work.	An empty file.	There should be an empty string read from the	An empty string was read from the file, it

			file.	contained nothing but the null terminator character.
U60	A yml file containing basic data types can be read.	A yml file containing basic data types, such as strings, integers and floats.	All three basic data types should be parsed correctly.	All three basic data types were parsed correctly.
U61	A yml file containing lists of data can be read.	A yml file containing lists of different data types.	Each list should be parsed correctly.	Each list was parsed correctly.
U62	An empty file should be correctly parsed by the reader.	An empty file.	The file should be parsed as a yml file with no data, and attempts to access data should throw exceptions.	The file was correctly parsed as an empty yml file, and attempts to access the data threw an exception.
U63	A yml file containing any arrangement of data should be parsed correctly.	A yml file with a typical looking set of data.	The file will be parsed correctly with all data accessible.	All data in the file was accessible when the yml file was parsed.
U64	A yml file containing syntax errors should be detected.	A yml file containing some syntax errors.	The file will not be parsed and an exception will be thrown.	The file was not parsed and an exception was thrown.

Test Plan – Video

Link to testing video: <https://www.youtube.com/watch?v=qdkdg5motuw>

QR Code to access the video can be found below. For convenience, a comment has been left below the video that allows each timestamp to be easily accessed.



Test Number	Test Purpose	Test Data	Expected	Actual	Timestamp
1	Make sure that all the unit tests pass.	Unit test.	All unit tests pass.	All unit tests passed.	0:19
2	Ensure that the simulation loads into a main menu.	Launch the program.	Main menu is displayed.	Main menu was displayed.	0:50
3	Make sure that text is being rendered correctly.	Large yellow text.	Splash text is displayed.	Splash text was displayed	1:01
4	//	Small white text.	Version text is displayed.	Version text was displayed.	1:03
5	Make sure that buttons in the gui work.	Left click.	Menu switches to builder menu.	Menu switched to builder menu.	1:12
6	Make sure that new objects can be created.	Left click.	A new object appears in the menu.	A new object appeared in the menu.	1:18
7	Make sure that the camera can be controlled.	The keys "W", "A", "S" and "D".	Camera should move around the world.	Camera moved around the world.	1:21
8	//	Right click, mouse movement.	Camera should rotate.	Camera rotated.	1:25
9	Make sure that objects are lit up by a light source.	Coordinate (0, 0, 10)	Different part of the object is lit.	Different part of the object was lit.	1:53
10	Make sure that	"cube.tmsh"	The new	The new	2:14

	models can be loaded from a file.		model should be loaded and displayed.	object was loaded and displayed.	
11	Make sure that materials can be loaded from a file.	"neon_orange.tmat"	The new material should be loaded and displayed.	The new material was loaded and displayed.	2:50
12	Make sure that multiple objects can exist in the world	Left click.	A second object should appear in the menu.	A second object appeared in the menu.	3:13
13	Check that position can be changed for an object	(5.0, 0.0, 0.0)	The object should move to the new position.	The object moved to the new position.	3:17
14	Check that velocity can be changed for an object.	(1.0, 0.0, 0.0)	The object should move with that velocity when the simulation (correct) is run.	The object moved with a speed of 1m/s.	3:53
15	Check that the acceleration due to gravity can be changed.	0ms^{-2}	Value should update and should be no acceleration downwards.	The value changes on the screen and no more gravity.	4:03
16	Pressing the run button will start the simulation.	Left click.	The simulation should change to show the objects in motion.	The simulation changed to show the objects are now in motion.	4:12
17	Should be able to change the name of an object.	"Orange"	Value should update in the editor and in the object hierarchy.	Value is updated in editor and in object hierarchy.	4:44
18	//	Boundary test – no input	Value should correctly	Value is updated	4:51

		given.	update, showing no name.	correctly and no name is shown.	
19	Erroneous test for entering non- numeric data in a numeric field.	"5.stupid"	Value should assume a default value of 0.0	Value is set to the default of 0.0	5:03
20	Erroneous test for entering non- boolean value in a boolean field.	"lksdfoijsof"	Value should be set to False by default.	Value is set to false by default.	5:18
21	Normal test for boolean input.	"true"	Value should be set to True.	Value is set to True.	5:26
22	Erroneous test for mass input, should not allow small or negative mass.	0	Value should be set to a small ish mass (0.0001)	Value is set to a small ish mass instead (0.0001)	5:47
23	Normal test for updating the mass.	234234	Value should be updated to the inputted mass.	Value was updated to the inputted mass.	5:58
24	Erroneous test for coefficient of restitution.	4560.5	Value should clamp to 1.0	Value is clamped to 1.0	6:09
25	Erroneous test for coefficient of restitution.	-3451	Value should clamp to 0.0	Value is clamped to 0.0	6:13
26	Erroneous test for file input.	"hello"	Should alert the user that there is no such file.	Alert saying "NOT FOUND"	6:30
27	Boundary test for file input.	"cube"	Should find a matching file for the model and use it.	Used the matching file "cube.tmsh"	6:39
28	Normal test for file input.	"cube.tmsh"	//	//	6:46
29	Make sure that	0.7, 0.0, 0.0,	Background	The	7:56

	the GUI elements can have their colour changed in a file.	1.0	colour of windows in the GUI should change to be red.	background colour of GUI windows became red.	
30	Make sure that GUI windows can be moved around the screen.	Left click mouse drag.	Window should move and cursor should change.	Window moved and cursor changed.	8:02
31	Ensure that windows can be easily modified to show different contents.	"Test new title"	The title of the save window should change.	The title is changed to the inputted value.	8:53
32	//	"DO NOT Press enter before saving."	The text in the window should change.	The text in the window is changed to the inputted data.	//
33	Test that scenarios can be saved.	"video"	The current scenario should be saved under the name video.	The current scenario was saved under the name "video".	10:14
34	Test that saved scenarios can be loaded from a save file.	"video"	The saved scenario should be loaded into the simulation.	The previously saved scenario was loaded into the simulation.	10:31
35	Check that simulation renderer is running independently of physics engine.	N/A	N/A	Running at ~144fps while the simulation is paused, showing they run independently.	11:48

36	Check that the simulation speed can be changed.	0.25	The simulation should run at one quarter of its original speed.	The simulation ran at one quarter of its original speed.	12:06
37	Check that the simulation is running with discrete time intervals.	Left click "Step" button.	The simulation should increment by one small interval of time.	The simulation advances by a single discrete interval of time.	12:22
38	Physics Question 1, find the speed of the ball when hitting the ground.	Heavy Ball simulation.	6.30ms^{-1}	6.28ms^{-1}	12:48
39	Physics Question 2, find the final speed after the cars crash.	Car Crash simulation.	5.00ms^{-1}	4.88ms^{-1} / 5.18ms^{-1}	15:15
40	Physics Question 3, find the speed of two identical masses colliding.	Perpendicular simulation.	7.1ms^{-1}	7.1ms^{-1}	17:00
41	Physics Question 4, student throwing a ball at the ground.	Throwing simulation.	7.25ms^{-1}	7.24ms^{-1}	19:31
42	Use of textures for objects.	N/A	Object surface should resemble an inputted image.	Object resembles a grass image as inputted.	20:18
43	Text window can be scrolled through.	Mouse scroll.	The later parts of the list is displayed but the order is maintained for the items.	The list scrolls as expected.	22:56

44	Check that physics rate can be changed.	30 updates per second.	The physics rate should change so that each step lasts a longer amount of time.	The physics rate is changed, the objects visibly move further between discrete steps.	25:00
45	Check that different types of exception are possible.	MEMORY_EXCEPTION	Should display the type of exception and the message a short message.	The correct message was displayed and the message was displayed.	27:23
46	Check that when an exception is called, execution should stop.	Functions + output to show function start/end.	The function that throws the exception should never display a terminating message.	The function that threw the exception did not display the termination message.	27:32
47	Ensure that when an exception is called, a stack trace is displayed.	Test functions.	The stack trace shows the order that the functions were called in to cause the exception.	The order of function calls was displayed correctly.	27:50
48	Check that different types of exception are possible.	NULL_PTR_EXCEPTION	Should display the type of exception and the message a short message.	The correct message was displayed and the message was displayed.	28:10
49	//	FAILURE	//	//	28:19
50	Check that changing the colour stored in a material will update the colour of an object.	(0.0, 1.0, 0.0)	The object should be displayed as green.	The object was displayed as green.	30:08
51	Ensure that each	N/A	N/A	The shaders	30:45

	material has an associated shader program.			are shown in the video, and how they are specified in the material file.	
52	Check that a material can have a texture associated with it that will change the appearance of an object.	"grass.png"	The object should resemble the grass texture image.	The object resembled the grass image.	31:25
53	//	"xavier.png"	The object should resemble the image.	The object resembled the image.	31:34
54	Check that the engine works using a fixed time step.	Left click (the step button)	The engine should advance in discrete steps of time.	The engine advanced in discrete intervals.	32:21
55	Check that the engine works with a fast rate.	100	The engine should run at 100 updates per second.	The engine ran without issue at 100 updates per second.	32:43
56	Ensure that the simulation follows Newtonian laws of physics.	(1.0, 0.0, 0.0) wait 1 second.	Distance = speed * time. Would expect to travel 1 metre if speed = 1ms^{-1} for 1 second.	The object travelled exactly 1 metre.	33:25
57	Ensure that a corrective impulse is applied to objects when they collide.	Two objects colliding.	An equal and opposite impulse is applied to each object to separate them.	Each object receives an impulse that changes their velocity by 0.25ms^{-1} , as expected.	34:15

58	Check that the GUI has text elements.	One of the menus.	There should be text rendered to the screen.	Text is rendered to the screen.	35:28
59	Check that the GUI has buttons.	The main menu.	There should be a button in the GUI.	There is a button in the GUI.	35:39
60	Check that the GUI has value sliders.	The builder menu.	There should be some value sliders in the GUI.	There were no value sliders present in the GUI.	35:46
61	Check that the GUI has tick boxes.	//	There should be a tick box (yes/no input) in the GUI.	There was a True/False input, but not a tick box.	35:50
62	Check that GUI elements are modular.	//	The GUI elements should be modular.	The GUI elements were determined to be modular.	36:05
63	Check that GUI elements can be combined together to make a full user interface.	//	The GUI elements should be combined together to make a full user interface.	Many GUI elements were seen on the screen to make a full user interface.	36:09
64	Check that the data stored in the GUI is easily accessible to the programmer.	Restitution value in the editor.	It should be easy to access the stored data.	It was easy to access the stored data, taking only a single function call and a human readable name (not an index for example).	36:28
65	Check that it is possible for	The GUI.	There should be labelled	There were many labelled	37:16

	labelled buttons to exist in the GUI.		buttons that perform actions.	buttons that performed actions.	
66	Check that each label in the GUI is concise.	//	There should be no labels that exceed 3 words in length.	There were immediately labels that were 4 words in length.	37:36
67	It should be possible to re run the simulation after making changes to the conditions.	5ms^{-2}	The acceleration due to gravity should change to 5ms^{-2} , and change should be reflected in the simulation.	The acceleration was changed and it was possible to re run the simulation with the new value.	37:58
68	There should be a button to allow time to flow in the simulation.	Left click (play button)	The simulation should run.	The simulation began to run.	38:14
69	There should be a button to pause the flow of time in the simulation.	Left click (pause button)	The simulation should pause.	The simulation was paused.	38:16
70	Check that there is a structure of oriented bounding boxes that act as the broad phase of the collision detection.	N/A	N/A	A short clip was displayed that shows the structure of the collision structure visually.	38:55
71	Check that using a broad and narrow phase collision detection reduces unnecessary computation.	Two objects colliding.	The number of checks required to determine whether objects are colliding should be	Checking each triangle would require 32400 checks to determine a non-collision, the optimised	40:21

			less than simply checking each triangle.	version needed a minimum of 0 and a maximum of 130.	
72	Check that collisions between two OBBs can be detected accurately.	Two OBBs not colliding.	No collision.	"There was not a collision"	43:18
73	//	Two OBBs colliding.	Collision.	"There was a collision"	44:01
74	//	Two OBBs colliding.	Collision.	"There was a collision"	44:38
75	Check that a collision between a triangle and an OBB can be detected accurately.	A triangle and an OBB not colliding.	No collision.	"There was not a collision"	46:20
76	//	A triangle and an OBB colliding.	Collision.	"There was a collision"	46:43
77	//	A triangle and an OBB colliding.	Collision.	"There was a collision"	47:04
78	Check that a collision between two triangles can be detected accurately.	Two triangles not colliding.	No collision.	"There was not a collision"	49:18
79	//	Two triangles colliding.	Collision.	"There was a collision"	49:41
80	//	Two triangles colliding.	Collision.	"There was a collision"	50:01

Evaluation

Objective Evaluation

Main Objective	Objective	Sub Objective	Evaluation	Test Number
Construct some programming utilities for the application.				
Create an exception handler.				
		It should be able to record different types of exception.	The program definitely is capable of handling many types of exception, however it is slightly annoying to add a new kind of exception, this could be improved.	45, 48, 49
		It should pause the current function if an exception occurs inside of it.	This works very well. This was an easy objective to achieve using a C macro.	46
		It should record a trace of all function calls that led up to the exception.	This works okay, there is a limit to the number of function calls that can be traced before the buffer is filled, but I never had issue with it.	47
Create a file utility.				
		It should be able to read data from files.	Works perfectly, however it needs another call to get the size of the data being read first. This is more due to limitations in the programming language used (C).	U57, U58, U59, 33
		It should be able to write data to files.	Works perfectly, as expected.	U57, U58, U59, 34
Create a YAML file utility.				
		It should be able to read a YAML file into a tree data structure.	This works for the time being. It does not have full support for YAML, there are some lesser used, shorthand ways to express	U60 - U64

			things that my parser cannot understand. For storing small configs and bits of data, where this shorthand isn't used often, there is no issue.	
		Each node of the tree should store the name of a key/identifier in the YAML file.	This was done exactly as planned.	U60 - U64
		Each node should have a child, which should either be another node of the tree, or data such as a string, number or list of values.	This was done exactly as planned.	U60 - U64
		There should be functions to access the stored data using the string keys.	There are functions, however I was not particularly happy with how it turned out. It required getting the YAML data in one call, and then calling another function to convert to the desired type afterwards. Once again, would not have even been an issue in Python, but a function that combines these into one would've been better.	U60 - U64

Construct a rendering engine for the application.

		The engine should perform a wide range of functions in two dimensions.		
		It should be able to render primitive shapes such as circles or polygons.	Was not used in the final program, despite the code being written. Zero impact for "missing" this objective.	N/A
		It should be able to render text at different font sizes.	The font engine can render font at any size, any orientation and any colour. Objective met with bonus features.	3, 4

		It should be able to render anti-aliased text.	The font renderer was not able to render anti-aliased text. I had never really used font that was not anti-aliased and expected it to be unreadable, but as it turned out to be perfectly legible without anti-aliasing I postponed this feature, and it never ended up being finished. As all text is readable still, I would argue this has a minimal impact on the final project.	N/A
	The engine should perform a wide range of functions in three dimensions.			
		It should be able to load the information needed to draw a three-dimensional shape from a file.	The program was capable of this. Works as planned. However, this is not particularly user-friendly, the files have to be written by hand (by keyboard?) in a custom format, meaning 3D models from other sources cannot easily be loaded. This could make the program less useable, so either a converter program or more advanced loader could be used.	10
		It should be able to draw a three dimensional representation of an object onto the screen, based on the position of the camera.	Works as expected.	9
	The engine should be able to perform lighting calculations.			
		It should be able to use different lighting effects that will vary the brightness of objects in the world based on light sources.	Works as expected, the shaders are not the most beautiful but they allow us to see the program running.	9
		It should be able to load different shader programs (programs that determine	Works as expected.	11

		the colouring of objects) from a file.		
	The engine should allow for different materials to be attached to objects.			
		Each material should have an associated shader program.	Implemented as stated.	51
		Each material should have an associated colour, stored as 3 floating point numbers representing the redness, greenness and blueness in a range from 0.0 to 1.0.	Implemented as stated.	50
		Each material should have (if required) an associated image file that serves to texture its object with a pattern instead of a flat colour.	Implemented as stated.	52, 53
		Each material should also have (if required) an associated image file that represents normal vectors on the face of an object in terms of the perpendicular x, y, and z coordinates. This will be used to change how lighting reacts to an object on different parts of a flat surface.	This feature was not implemented. The primary focus on the project was to get the physics simulation working, and time constraints meant that I did not get chance to work on the rendering engine as much as I would have liked. While this would've allowed for more interesting looking objects, things like scratched metals, wet rocks or having shiny parts of an otherwise dull object, this would have had no effect on the physics. Therefore, there is a minimal impact on not having this feature.	N/A
Construct a physics engine.				
	The engine should accurately depict real-life physics.			
		The engine should work by simulating physics in small	Implemented as stated. Small intervals are passed	54

		intervals of time.	in each iteration of the engine loop.	
		The engine should use a fixed time interval, allowing it to perform the same calculations regardless of hardware speed and other factors.	The engine used a fixed time step. The way I implemented this was to wait a certain time between each iteration of the main loop. I think that the for loop should not have waited, it should be continuously checking the event queue for messages. If on a particular loop it is detected that the time step has passed, then a physics update should occur. The lack of this feature means that a low update rate causes an unresponsive experience. During typical use (100Hz loop), the effect is not noticeable, and the simulation is not useful at slow refresh rates anyway. Therefore there is arguably only a small impact on the final program.	54
		The engine should use equations from Newtonian physics to predict the motion of objects between time intervals.	There was no direct usage of the Newtonian equations such as SUVAT, these were all encoded by the velocity integrator and the collision resolver. However, the same answer will be reached. As a simulator, not a solver, this is to be expected.	56
		The engine should use respond to colliding objects by applying a corrective impulse to displace the colliding objects.	This was implemented, the impulse was not a change in momentum but rather a change in velocity that was added.	57
	The engine should work in real-time			
		The engine should use a	The engine used a time	55

		time interval of less than 0.05 seconds (20 intervals per second).	interval of 100 per second, or 0.01 seconds per update.	
		The physics engine should run independently of the rendering engine, allowing it to run at the correct pace even if the rendering is slowed.	The physics engine and graphics ran on separate threads, this allowed them to remain independent. The main issue would be graphics slowing rather than physics as the graphics are very simple.	35
The engine should be optimised for fast running				
		The engine should use a tree of oriented bounding boxes that contain objects in increasingly small containers, so that collisions can be checked using a binary search rather than testing each face individually.	This section was really the key focus of this entire project, and constitutes a great deal of the code and complexity. I would argue that this section was completed exactly as stated in the objectives. There are possibly a few optimisations left to make, some calculations could be cached between physics steps for example, but that was never in the objectives.	70
		The engine should use a well-tested triangle collision algorithm such as GJK + EPA	//	78 – 80
		The engine should employ the separating axis theorem to test collisions between oriented bounding boxes.	//	72 – 77
		The engine should use a broad and narrow collision phase to avoid unnecessary computation.	//	71
Construct a user interface to allow users to control the program				
	It should be easy to program new areas of the user interface (UI)			

		<p>There should be UI elements such as containers, text, buttons, value sliders and tick boxes.</p>	<p>There were text and buttons implemented. There were no tick boxes, but there were true/false text entries which function the same. There were no value sliders, these may have been easier to use for values that have a range, for example the coefficients of friction and restitution are always between 0.0 and 1.0. Overall, there is a small impact on user friendliness, but no impact on actual capability of the program.</p>	58 – 61
		<p>UI elements should be modular, meaning they can be combined together to make a full user interface.</p>	<p>The UI elements were coded as stand-alone parts, e.g. a text button, a text input, a window. These were combined to make more complex structures such as the editor windows, the main menus. Therefore, the UI elements were combined to make the full user interface. I had envisioned something similar to html <div> tags, that would allow different elements to resize based on their parent container, but the UI never became complex enough to require this.</p>	62, 63
		<p>Interactive UI elements should allow the programmer to access their stored values.</p>	<p>Implemented as expected.</p>	64
		<p>It should be possible to store a UI configuration in a file, so that the UI can be modified without rewriting any code.</p>	<p>Implemented as expected. There is a file to set the colour and other properties of elements, and there are also windows that can have their layout specified by a</p>	29, 31, 32

			file (option windows).	
	It should be simple for the user to navigate the UI			
	The UI should have labelled buttons that can be pressed to perform different actions within the simulator.	Implemented as stated.	2, 5, 6, 12, 16, 37, 54, 65	
	The UI should have short labels (less than 3 words) for buttons or other inputs, to help keep the UI concise.	There are some labels that have more than 3 words, for example "acceleration due to gravity". Arguably, this could be rewritten as "gravitational acceleration", though arguably the reduction of those 2 words does not make that label any more readable. While the target of 3 words is good to help keep the menus concise, it would be unfair to penalise this slip into 4 words. No impact on "missing" this objective.	66	
	The UI should contain help buttons that give a description of the functions of each button or input.	No such help button was conceived. The menu was not particularly complex or hard to understand, but the program may be difficult to understand without being told what to do. Small impact from missing this objective.	N/A	
	The UI should change relevant to the state of the simulations			
	The UI should have a main menu that allows users to select a simulation mode (pre-existing scenarios, or a scenario builder) or to edit the settings	The main menu did not have the option to load pre-existing scenarios, as none were made. One of the main ideas of this program was being able to play with some pre-existing scenarios and work on them. Medium impact.	N/A	

		The UI should be able to change to a settings panel when the settings button is pressed, which will allow the user to configure the graphics settings and physics simulation settings.	There was no dedicated settings panel to edit, however the settings were available through other means, either by editing external files or using other sections of the UI. No / negligible impact.	N/A
		The UI should be able to change when the scenario builder mode is activated, to allow scenarios to be created or updated	Implemented as expected.	5
		The UI should be able to change when the simulation is being run, to display the simulation running in real time.	Implemented as expected.	16
The UI should be effective during the creation of new scenarios				
		The UI should allow users to quickly edit objects' masses, initial velocities, constants such as friction and restitution, colour and shape.	All could be edited bar colour. The colour could not be edited through the editor, but a material could be chosen that has a different colour. No impact.	9 – 11, 13, 14, 17 – 28
		The UI should be operable using the mouse to drag objects or points on objects to edit them.	The mouse could not be used to move objects in the scene. Instead, the position had to be edited directly by specifying a coordinate. This made the program unintuitive to use and hard to make changes to the scene. High impact on user experience.	N/A
		The UI should allow for changes to be undone and redone, making use of buttons or the shortcuts Ctrl-Z and Ctrl-Y respectively.	The undo/redo shortcuts did not work due to not being implemented; this could have potentially devastating impacts on projects if accidental changes are made. High impact on user	N/A

			experience.	
		The UI should allow objects to be copied, cut, and paste in the world, making use of buttons or the shortcuts Ctrl-C, Ctrl-X, and Ctrl-V respectively.	The shortcuts for cut, copy and paste did not work. This means that if identical objects are needed, then you have to manually input the data twice. High impact on user experience.	N/A
		The UI should allow the simulation to be re-run following any changes to the starting conditions.	Implemented as expected.	67
The UI should be effective during the running of scenarios				
		The UI should display time control functions such as play and pause, which will allow and block the passage of simulation time.	This was a really useful feature that made the program really interesting to use. No issues, works as expected. Possibly could use a single button that flips between pause and play instead of two separate buttons.	68, 69
		The UI should display time control functions that change the speed at which time advances, so that physics can be viewed in slow motion.	This was also a very interesting feature, it allowed for fine-grained control over the simulation. This area could possibly have benefitted from the "slider" UI element, instead of typing the speed in each time. Functionally works as expected.	36
		The UI should also allow time to be experienced in reverse, by caching the states of objects at a steady time interval and then displaying their progression in reverse chronological order when requested.	This feature was not implemented. This would've been interesting to get to repeat a specific portion of the simulation when wanted. High impact on program usefulness.	N/A
		The UI should have the	This feature was not	N/A

		option of exporting simulated data as a spreadsheet file (.csv) for further processing if the user needs the raw data.	implemented. This would have allowed for further processing of the data in other applications such as excel to draw and study graphs of motion during a collision. High impact on program usefulness.	
		The UI should be able to display graphs such as velocity-time graphs that represent the motion of different objects.	This feature was not implemented. High impact on program usefulness.	N/A

End User Feedback

Interview

Following the completion of the project, I interviewed my end user again to see what they thought of the end result. I asked them a few questions on what they thought, liked or disliked about the program.

After looking at the simulator, what are your initial thoughts of the program?

So, first impressions - honestly, I'm pretty impressed. The fact it's all in 3D is a big step up from what I was expecting. Most of the old software we used to have was very... flat, essentially. Being able to actually see objects moving around in space, colliding, doing what they should be doing - it makes it feel a lot closer to the real thing. And the menu system you've put in, where you can type in vectors directly, that's really nice. That's the sort of thing that makes my life easier because if I know something should be, say, (3, -1) or whatever, I can just type it in and not fiddle around guessing.

And the fact that you can just add objects, hit run, and it behaves exactly as the physics says it should - that's the bit I like. That's the bit where, in a classroom, I could point at something on the board, recreate it quickly, and say "look, this is what actually happens." So as a first go, as a working prototype, yeah - really good. There are things missing, of course, but what's here is solid.

How much do you think this program could help you to explain different concepts in physics on a scale of useless to game-changing?

Hmm... okay, if we're putting it on a scale like that - useless at one end and game-changing at the other - I'd put this comfortably in the "very useful heading toward game-changing" area.

The reason is, when you're teaching mechanics, half the battle is getting students to picture what's going on. They can recite $F=ma$, fine, but actually

seeing why something accelerates the way it does - that's the tricky bit. And normally I'm there waving my hands around, grabbing pens and rulers to pretend they're forces, all that sort of thing. This gives me a way to show it properly, cleanly, and consistently.

Now, without the force arrows and the graphs, I wouldn't quite call it game-changing yet, because those are the bits that let you say "look, here's the force, here's the acceleration, here's how they relate." But even in its current state, just being able to build a scenario, run it, slow it down, all that - that's already extremely helpful.

So yeah, definitely far closer to the game-changing end than the useless end. It's got real potential.

Which feature are you most happy about in the program?

Oh, that's easy - the real-time simulation with adjustable speed. That's the one that really makes me go "yes, this is what we've been missing."

Being able to set something up, press run, and then slow it right down so you can watch exactly what's happening... that's enormously helpful. In real life everything happens too fast - collisions, changes in direction, acceleration building up - you blink and it's done. But here I can say, "right, look at this bit, this moment here," and the students can actually see the motion rather than just trusting the algebra.

Which feature would you like to see added to the program if I were to continue development and why?

If I had to pick just one to prioritise, I'd go for force arrows, without a doubt.

The reason is: that's the missing link between seeing what happens and understanding why it happens. At the moment the motion is correct - which is great - but students often struggle to connect that motion back to the forces that caused it. If I could pause the simulation and say, "Right, here's the weight, here's the normal reaction, here's the horizontal force," and actually point to arrows on the object, that would turn it from a demonstration tool into a proper teaching tool.

And ideally the arrows would update in real time - so if something starts accelerating, the resultant force arrow changes with it. That kind of visualisation is gold dust in mechanics, because it makes the invisible visible.

If I were to go back and restart this project, would you want to see any fundamental changes to how the program works, or is this concept really solid?

No, I don't think you need to tear anything up and start again - the underlying concept is absolutely solid.

The core idea - build a scenario, set the vectors, press run, and watch it behave according to proper physics - that's exactly what a tool like this should do. You've got the foundations right. The 3D environment, the ability to input

vectors directly, the real-time simulation... all of that is the "engine room," and you've clearly got that working well.

So no, the fundamentals are spot on. If you restarted, I'd want you to build the same thing again - just with more time to polish the extra features you didn't get chance to add.

Interview Evaluation

Following the interview, I decided to evaluate the feedback that I was given. To start with, the feedback is quite realistic. I could definitely see some simple scenarios being displayed using this program to a class, even if just to put the image of how it would look into their heads while they're working out the problem, and they could see how the numbers relate to how it looks. I would also say it's quite fair to the project, my user didn't expect there to be any additional features that weren't originally asked for, and the ones that I managed to implement were appreciated. The feedback is definitely representative of my project, the parts that I spent the longest on – the collision detections, the core physics engine – these all stood out in the end interview. The bits that I left until the end – the mouse controls, displaying force arrows – these bits were picked up in the interview as lacking. Obviously, it's not good to hear that your project didn't quite meet the expectations, but I don't feel surprised that this is the outcome.

Future Improvements

Air Resistance Simulation

One way to improve the solution could be to include air resistance in the calculation of object velocities. I think this would be beneficial to people looking to simulate things that would happen in real life rather than modelling simple exam questions in the classroom. For example, people who are interested in activities like sky diving or base jumping could predict the best time to open their parachute by testing a model with their mass and a large parachute shape to the simulation, and testing different starting heights to find which ones allow them to slow down enough. However, I would not be taking responsibility for accidents caused if the predictions were inaccurate.

There are two ways that I could think to implement air resistance to my simulation. The simple way would be to project the object onto a plane perpendicular to the direction of travel. Then calculate the area of the projected polygon on this plane. You could then make a rough estimate of air resistance as the force of air resistance is proportional to the cross-sectional area. However, this would assume that a flat face and a pointed face have the same air resistance, this is not particularly accurate.

You could improve this method by giving each object an "air resistance" value, we see this approach being taken in the game "Trailmakers", where each building block for a creation has a different air resistance – for example cones and

wedges have a lower value than flat blocks. There is a value for the air resistance in 6 directions, which are the positive and negative direction in each axis. This is still relatively simple to implement. A rough sketch of the code in pseudocode to achieve the air resistance is shown below.

```

FUNCTION DoAirResistance(body)
    // assuming that:
    // +X = 0
    // -X = 1
    // +Y = 2
    // -Y = 3
    // +Z = 4
    // -Z = 5
    x_selector ← 0
    y_selector ← 2
    z_selector ← 4
    IF body.velocity.x < 0 THEN
        x_selector ← 1
    ENDIF
    IF body.velocity.y < 0 THEN
        y_selector ← 3
    ENDIF
    IF body.velocity.z < 0 THEN
        z_selector ← 5
    ENDIF

    total_v ← body.velocity.x + body.velocity.y + body.velocity.z
    x_v ← body.velocity.x / total_v
    y_v ← body.velocity.y / total_v
    z_v ← body.velocity.z / total_v

    air_k ← body.air_resistance[x_selector] * x_v +
    body.air_resistance[y_selector] * y_v +
    body.air_resistance[z_selector] * z_v
    air_resistance ← air_k * MAGNITUDE_SQUARED(body.air_resistance)

    ApplyForce(body, air_resistance)
ENDFUNCTION

```

You would also need to add to the model files this air resistance data, or otherwise find a way to calculate it based on the shape of the model.

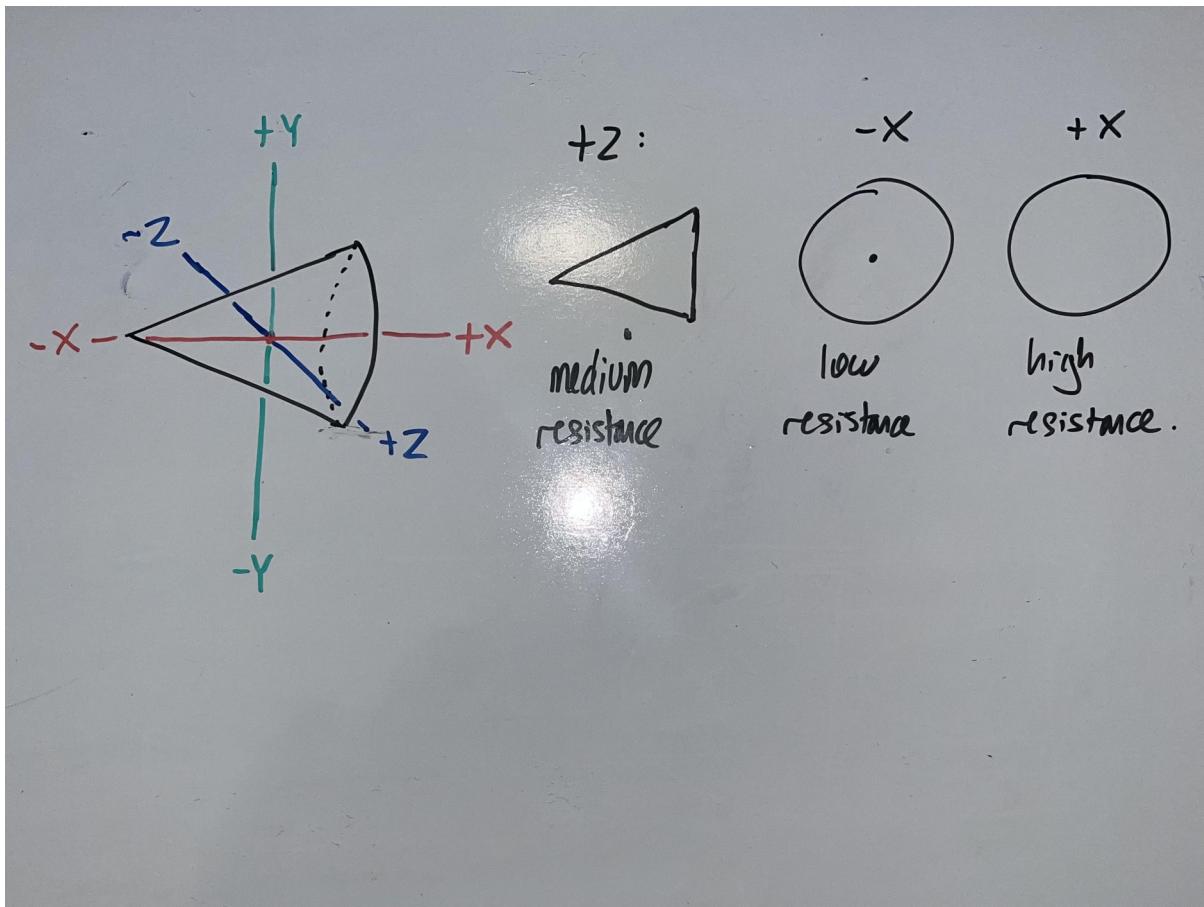


Figure 35: A diagram showing how air resistance could be implemented on a cone shape.

The other way to implement this would be a lot more complicated, but you could use computational fluid dynamics. Then, you can divide the world into finite elements or finite volumes to simulate whichever fluid you require. Then, there is an iterative solver step (a similar concept to the iterative constraint solver used in TekPhysics already) which computes the velocity, temperature and pressure of the fluid in each finite cell. However, it may be more challenging to implement when there are multiple moving objects that all disrupt the flow of fluid.

Increased Model Customisation

It may be useful to allow users to customise the models they use to allow for more scenarios to be simulated. For example, the user could specify a different centre of mass if they do not want to simulate an object with constant density. This could be implemented quite easily with some adjustments to the ".tmsh" file loader to search for a tag that specifies the centre of mass. Then, if a custom centre of mass is provided by the user, the calculation for it can be skipped.

Another customisation ability could be to allow for objects made of multiple materials, for example a wheel with a rubber tyre and a metal rim. This could allow more realistic objects to be added to the game, as there are actually very

few objects that are a continuous block of the same material apart from construction materials and such.

The first way I could think to implement this would be to use the existing model loader, load a different model for each material used, and then use the constraint solver to enforce that each model is connected to each other like a single object. This would probably require the least amount of changes throughout the code, as only an extra constraint equation is needed, however it would probably be quite tedious to get working and could have issues with forces not propagating throughout the object as you would expect – looking at the tyre and rim example, a force to the tyre may produce a larger than expected acceleration as it may not take into account the total mass of the rim, due to it acting somewhat independently.

The second way I can think to implement this would be to overhaul the model loader, and allow each polygon to have its own friction coefficient, restitution and density. This would mean that the object could be modelled as a single body, although a lot of code changes would be needed. Firstly, the renderer would need to be able to switch between different lighting and textures for each material in the model. The collision response algorithm would also need to be modified to use the friction coefficient per polygon instead of per object. This shouldn't be too difficult to achieve, it would be just a matter of updating the collision manifold. I would also need to consider how each polygon would get these properties, whether to duplicate the properties for each polygon, or splitting the mesh file into a section for each material. However, I think that despite these complexities, this approach may end up being more stable during the actual simulation.

Simple Scripting Abilities

The simulation could be improved by allowing users access to a scripting service in a language like Lua or Javascript. This would be useful as it could allow for much more complicated problems to be solved, perhaps something like "A rocket falls from a height of 30km and fires its thrusters at a height of 500m above the surface, will it collide with the ground before it reaches 0 velocity?". Then, the user could add a simple script that looks something like this:

```
1 FUNCTION OnUpdate(state)
2     rocket ← state.GetObjectByName("Rocket")
3     IF rocketGetPosition().y <= 500
4         force ← Vec3(0, 1000, 0) // Assuming the force is 1000N
upwards.
5         rocket.ApplyForce(force)
6     ENDIF
7 ENDFUNCTION
```

This could be implemented by including a language interpreter in the solution. The user could then specify a script for the object in their object files, which the solution could load with the object. Each script could contain methods like "on load", "on update", "on collision". Then, on each physics update, all the scripts are called for each object that has a script attached. Now, each script would be passed a data structure or class containing all the objects in the world, and some methods to retrieve the object they would like to modify. They could access a dummy object that has all the current values for mass, velocity, and so on for that object, which they can use different methods to modify as they please. After the function terminates, the dummy values are relayed back to the main script and they will be applied to the real object.

Appendices

1. Finding the inertia tensor of a solid body:
http://number-none.com/blow/inertia/body_i.html
2. Finding the inertia tensor of a tetrahedron:
<https://docsdrive.com/pdfs/sciencepublications/jmssp/2005/8-11.pdf>
3. Separating axis theorem for two oriented bounding boxes:
<https://jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf>
4. Collision test for oriented bounding box and a triangle:
<https://gdbooks.gitbooks.io/3dcollisions/content/Chapter4/aabb-triangle.html>
5. Allen Chou's game physics series was immeasurably helpful for this project: <https://allenchou.net/game-physics-series/>
6. This video by Casey Muratori massively helped with writing my GJK triangle collision algorithm: <https://www.youtube.com/watch?v=Qupqu1xe7Io>
7. Finding contact points of triangles using EPA:
<https://winter.dev/articles/epa-algorithm>
8. I have been using this resource to learn OpenGL for the past 5 years now, most OpenGL I wrote from memory but it would have originally come from here: <https://learnopengl.com/>
9. Generating a sphere from triangles:
https://www.songho.ca/opengl/gl_sphere.html