Project Report

Graphics

Implementation of Missing Subsystem: OpenGL Display Engine

I have completely replaced the engine's display system with OpenTK, a C# that encapsulates OpenGL. Additionally, OpenTK provides useful Math functionality like vectors, matrices and quaternions, which we use in various places. I integrated it into the existing game loop by calling a preDraw event where the display engine prepares for a new frame to be rendered. I extended the game loop by a draw event for the running game and a draw event that fires for every gameobject. Subsequent graphics features build on this system and are aimed at making drawing easier by encapsulating commonly used functionality.

Group Members responsible: Pascal (100%)

Where it can be found

DisplayOpenGL.cs, WindowGL.cs, changes to Bootstrap.cs

Grading Expectation

I believe this to be a major rewrite of a substantial subsystem of the engine and would estimate this to be worth 15 points.

Justification

OpenGL is a powerful display engine. The hard part of this implementation was the integration into the game loop, making sure window updates etc. fired at the right time. This required a deep understanding of the existing OpenTK systems.

Moderate Extension to the Engine: Shaders

I added a class to encapsulate GLSL shader programs, allowing the user to simply load a shader program from a file and skipping low-level steps like compiling and linking. I also provide shortcuts to apply loaded shaders, abstracting away the use of OpenGL handles. Lastly, I implement a basic default shader that renders models with simple pre-defined directional lighting, taking into account the currently set model, view and projection transformations and provide a static function in Shader.cs that loads and sets this default shader.

Group Members responsible: Pascal (100%)

Where it can be found

Shader.cs, Shaders/default.vert, Shaders/default.frag

Grading Expectation

I believe this to be a moderate extension to the engine of ~10 points.

Justification

Shaders are absolutely necessary when rendering anything with OpenGL. However, many users find working with shaders to be very confusing so I see a high value in abstracting them, especially in simple use cases. The work here is in properly handling shaders, i.e. loading, compiling, linking and disposing of them correctly. Some time was also invested in a usable default shader.

Moderate Extension to the Engine: Camera

The Camera class - which is a GameObject subclass - abstracts the use of the projection and view matrices within the OpenGL display engine. The user sets an FoV-angle, near and far clipping plane. Position and orientation of the camera are set through the gameobject's Trransform3DNew. Then the user can get the camera's view and projection matrices by simple function calls or simply set it as the displayEngine's main camera, setting the matrices automatically.

Group Members responsible: Pascal (100%)

Where it can be found

Camera.cs, DisplayOpenGL.cs

Grading expectation

I believe this to be a moderate extension to the engine of ~10 points.

Justification

Working with matrices and transformations is challenging to many users, while the concept of a camera in 3D space is much more intuitive, which is where I see the value of this extension. The work here is in correctly building the view and projection matrices from the values the user provides.

Major Extension to the Engine: Meshes

OpenGL represents 3D data as vertex buffers, which can be a hassle to set up and work with. To make this easier I introduce the mesh class that abstracts loading 3D meshes into a vertex buffer and drawing them. Meshes can be loaded in 3 different ways: (1) Giving no arguments to the constructor loads a simple predefined unit quad mesh. (2) Vertex and index arrays can be given to the constructor to load arbitrary meshes. (3) Modeling tools like blender can export .obj files containing vertex positions, normals and texture coordinates. The ObjLoader class provides functionality to read these files and instantiates a mesh object holding the appropriate data. However, this is currently only able to read triangulated meshes, ignores any other data than position, normals and texcoords and is not very memory efficient. The vertex buffer format the mesh class uses is compatible with the default shader introduced earlier.

Group Members responsible: Pascal (100%)

Where it can be found

Mesh.cs, ObjLoader.cs

Grading expectation

I believe this to be a major extension to the engine of ~15 points.

Justification

Abstracting away vertex buffers increases ease of use significantly. Also, the ability to load .obj files improves workflow with 3D software drastically. The work here was put in properly handling 3D data and establishing a proper vertex buffer format. Additionally, loading functionality for .obj files took research into the structure of these files and a lot of debugging.

Minor Extension to the Engine: Textures

The texture class provides abstracted functionality to load, configure and apply textures. I use the StbImage library to load image files. Default texture wrap and filtering modes, that should work well enough in most cases, are set to allow users to simply use textures without thinking about technical details.

Group Members responsible: Pascal (100%)

Where it can be found

Texture.cs

Grading expectation

I believe this to be a minor extension to the engine of ~5 points.

Justification

Textures are powerful tools in 3D rendering. Here I had to put some work into getting image files to load properly and correctly handling the technicalities of textures in OpenGL.

Moderate Extension to the Engine: Animated Meshes

The AnimatedMesh class - a Mesh subclass - facilitates a mesh with an animated texture. This would mostly be used to render billboard objects like in early 3D games (e.g. enemies in Doom), which we also use in our demo game. Additionally to the mesh itself, an animated mesh holds a texture containing a horizontally laid out sprite sheet. The user specifies how many frames the animation contains, at what speed it should play and whether or not it should loop. The class provides a function to start the playback of the animation and determines the correct frame to display at each draw call. It then passes on this information to the shader animated shader, which determines what part of the sprite sheet to display on

the object. Loading functionality for this shader is provided in the shader class. A function has been added to the Mesh class allowing their conversion to animated meshes.

Group Members responsible: Pascal (100%)

Where it can be found

AnimatedMesh.cs, Shaders/animated.frag, Shader.cs, Mesh.cs

Grading expectation

I believe this to be a moderate extension to the engine of ~10 points.

Justification

Early 3D games make common use of animated textures. This is a simple yet efficient implementation of this feature. Using only one sprite sheet prevents loading individual textures for every frame. The work here is in determining the correct frame to display according to all parameters set by the user and having the shader calculate the correct region of the sprite sheet to display.

Minor Extension to the Engine: Physics Debug

Similar to the way colliders were displayed with the SDL display system and 2D colliders, I implement the same functionality for the new OpenGL display system and 3D colliders. DrawMe functions are provided for cube and sphere colliders and are called from PhysicsBody. Sphere colliders still draw their AABB instead of a proper sphere.

Group Members responsible: Pascal (100%)

Where it can be found

ColliderCube.cs, ColliderSphere.cs, Shader.cs, Shaders/wireframe.vert, Shaders/wireframe.frag, small adjustment to PhysicsBody.cs

Grading expectation

I believe this to be a minor extension to the engine of ~5 points.

Justification

This was relatively straightforward to implement but serves as an incredibly useful debug tool. The work here was in correctly specifying a vertex buffer to represent lines making up the bounding box of the collider. Since these vertex buffers use a different format a new wireframe shader was necessary to display them correctly. Analogous to the default and animation shaders, setup functionality has been added to the shader class.

Collisions

Major Rewrite of Existing Engine Feature: Transform

The original Transform3D class provided some 3D functionality. However, we found it to be inconsistent and unsuitable for our use case due to its strong tie to a 2D environment and sprite based graphics. We replaced the original system with the Transform3DNew class. It represents an object's 3D position, rotation and scale by a vector, quaternion and vector respectively. We provide functions to translate and rotate an object, as well as to retrieve matrices representing the (inverse) transformation from the origin described by the transform.

Group Members responsible: Jon (50%), Pascal (50%)

Where it can be found

Transform3DNew.cs, every class that uses Transforms and has not been replaced

Grading expectation

We believe this to be a major extension to the engine of ~15 points.

Justification

3D graphics rely heavily on matrix transformations, which the old implementation did not support. We found it more sensible to rewrite the system than to adapt the original implementation, which was primarily aimed at a 2D, sprite-based context. The work here was in establishing a system that both works well with the new display system as well as the collision system. Implementing the functionality required some research about matrix math etc. Lastly, we had to adapt the rest of the engine to fit this new system, which required a substantial amount of debugging.

Minor Rewrite of Existing Engine Feature: Collision Checking

Making the original collision detection system work with our 3D colliders required some adaptions and rewriting of the board and narrow pass. Ideally, for 3D scenes, the broad pass would happen along 2 axes. We ignored this for now and just made the current system work with 3D colliders. This has not posed a performance problem so far there were attempts at making

Group Members responsible: Jon (50%), Pascal (50%)

Where it can be found

PhysicsManager.cs

Grading expectation

We believe this to be a minor extension to the engine of ~5 points.

Justification

It took some time to fully understand the broad and narrow pass system and some debugging to finally make it work with the new 3D colliders.

Major Extension to the Engine: Collision Prevention with Stationary Objects

We had trouble using the original collision detection system to facilitate collisions of moving objects with stationary objects, where the moving object should be completely stopped by the stationary one and not be able to move through it, e.g. the player in an FPS game walking against a wall. The original system did not prevent objects from moving through one another. It also does not allow to easily check for collisions at the next position of an object before actually moving it. That is why we decided to implement a grid structure holding all stationary objects (i.e. those with the kinematic flag set to true). Each grid cell holds a list of all stationary objects that intersect it. We provide functionality to query this grid by giving a collider and an offset. We then return whether or not the collider intersects any stationary object. The offset is used to offset the collider by its PhysicsBody's velocity to query the position it will be at in the next frame. If a collision in the next frame is detected, movement will be stopped before a collision occurs, successfully preventing PhysicsBody's from clipping into stationary objects like walls.

Group Members responsible: Jon (20%), Pascal (80%)

Where it can be found

PhysicsManager.cs, PhysicsBody.cs, ColliderCube.cs, ColliderSphere.cs

Grading expectation

We believe this to be a major extension to the engine of ~15 points.

Justification

To arrive at this solution we tried several methods using the original system without success. Implementing the grid structure and the functionality to fill it with data and query it required a fair amount of debugging. Adjustments to ColliderCube and ColliderSphere had to be made in order to allow for an offset before performing a collision check.

Major Extension to the Engine: 3D Colliders with Cubes and Spheres.

As the original collision detection system only allowed for 2D collisions and there being reliances on 2D vectors for the abstract class Collider.cs. It was necessary to add 3 new classes which could accommodate 3D colliders. This took longer than expected due to mostly how integrated certain parts were with the current system. The physicsmanager.cs currently and all it handled was very dependent on the usage of 2D vectors and the Collider.cs class. This meant that all of this had to be accommodated as well. While the

collisions ended up being pretty basic, it now only returns a boolean whether or not a collision was detected, attempts were made to make it similar to the current 2D system by returning a vector. However, no easy way was found to do this as it gets more complicated in 3D and it would require an even further rewrite of the physics engine for it to work properly. Still a lot of time was spent on this even though the efforts were futile. Currently the functionality works as follows, the ColliderSphere Calculates a bounding box depending on the physics body's current position given a radius. The ColliderCube works similarly but requires height, depth, and width.

Group Members responsible: Jon (80%), Pascal (20%)

Where it can be found

ColliderCube.cs, ColliderSphere.cs, Collider3D.cs, PhysicsManager.cs

Grading expectation

We believe this to be a Major extension to the engine of ~15 points.

Justification

Because of the time spent trying to implement further functionality than just a simple boolean colliding box, the efforts were more than what it seemed, further, the classes are still relatively big therefore we believe it to be more than just a moderate extension and more in line with a major extension. Also the system depends on this working so a lot of time was spent making sure that everything in this class works in accordance with the rest of the implementations.

Demo Game: Basic Doom

A game was also made to showcase these features. We decided to implement a first person shooter game where the goal is to shoot enemies and survive as long as possible. The monsters chase are dumb and chase after the player's current position until they reach you, meaning they will run into walls as well. Shooting a monster kills it, a new monster then respawns in a random location. Over time, the game increases in difficulty by spawning multiple monsters at the same time and making them move faster.
The game showcases all the features mentioned above. An annotated screencast gives more information on how we use the engine features we implemented for the game.

Group Members responsible: Jon (60%), Pascal (40%)

Where it can be found

GameGLTest.cs, Player.cs, Bullet.cs, Monster.cs, Wall.cs

Grading expectation

We think this game is not yet 100% feature complete as it lacks e.g. a game over screen, but it is more than a simple showcase of our engine features making it fit somewhere in between. We estimate it to be worth ~25 points.