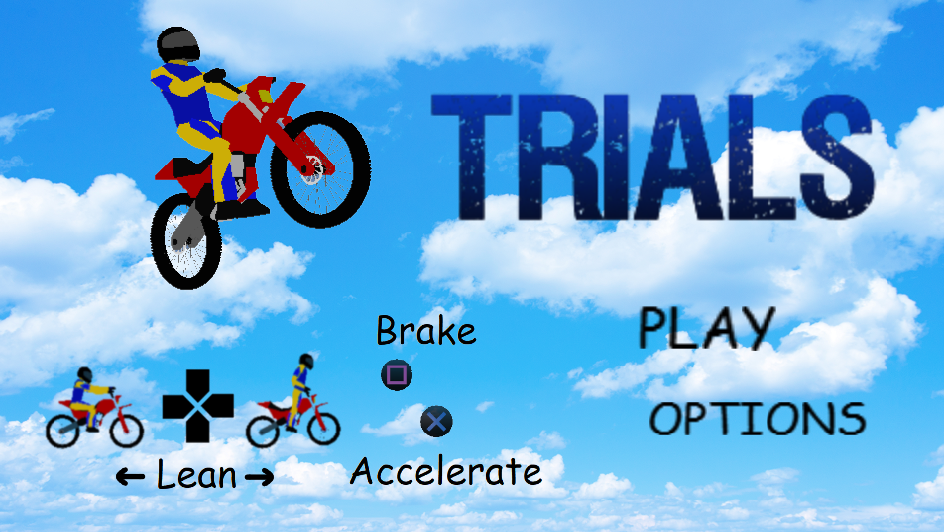
CMP 208 - Coursework

Introduction

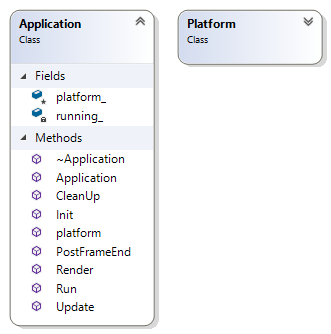
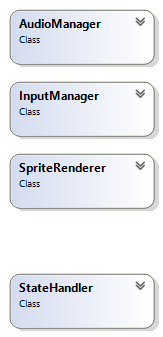
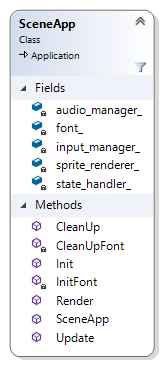
The application is a dirt bike simulation inspired by the “Trials” series by Ubisoft. It is a so called 2.5D game (3D graphics in a 2D gameplay environment). It is created for the PlayStation VITA using the GEF (Games Education Framework) library. Box2D is the physics library chosen for this application. It is written in C++ using Visual Studio as the IDE (Integrated Development Environment).

The aim of the game is to navigate the track in the shortest possible time without falling. There are various obstacles you have to traverse. You will need to use the rider’s body weight in combination with the accelerator and brake to overcome these obstacles. The idea is to keep the rider balanced on the bike by leaning backwards and forwards. The bike has front and back suspension that you can use to your advantage as well.

If any part of the rider comes into contact with the ground or an obstacle, you will lose your grip and fall off the bike. If you crash you will have the option to restart the level or go back to the main menu. If you reach the end of a level, you can continue on to the next one. There is also an options menu where you can select what level you want to attempt. Currently all levels are unlocked but for release you would have to complete the previous level before being able to select the next one.



Application Design

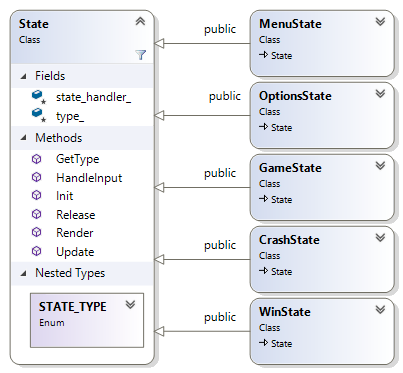
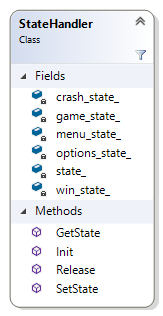
As the application is quite large, there is a need for the structure of the software to be well thought out. An Object Oriented approach was adopted for this design.

Everything starts in SceneApp that extends from Application. Application has an instance of Platform that handles shaders. This is encapsulated so that the developer does not need to worry about this. The underlying framework calls SceneApp’s Update and Render functions every frame.

SceneApp’s update function calls StateHandler’s Update function that calls the current State’s Update function. The first thing the State’s Update function does is call the State’s Handle Input function. It then goes on to update any variables and call other State specific functions to execute desired functionality. SceneApp’s Render function calls the current State’s Render function. This creates a standard game loop of handle input, update and render. This helps to keep code organised and readable. There is limited code in the State’s Update functions by design as this makes the code easier to read. Instead it is encapsulated in the class to which it belongs where possible.

SceneApp creates instances of AudioManager, InputManager and SpriteRenderer. These classes handle various functionalities that the developer can use. They are declared in the GEF namespace. SceneApp passes pointers to these instances into subsequent classes so they can access them.

Initially states were declared and changed in SceneApp as all the available states needed to be accessible. It was impossible to change states from inside the child classes of State, as the child classes could not see one another. This meant input to handle changing states had to be handled in SceneApp and only the state specific input was handled in the State’s child classes. It seemed inappropriate to have input handled in more than one place. It was also potentially dangerous as input conflicts could emerge, producing unexpected behaviour. The solution was to create the StateHandler class. SceneApp creates an instance of StateHandler.



StateHandler has the sole purpose of dictating what state the application is in. It has an enumerator to indicate the current state. StateHandler creates instances of MenuState, OptionsState, GameState, CrashState and WinState. These are all child classes of the base class State. It has a State pointer that points to the state the application is currently in.

It has a SetState function that takes an enumerator as an argument that indicates the desired state to change to. This function calls the CleanUp function of the current state (which deletes any objects declared on the heap), sets the pointer to the memory location of the desired state and calls the Init function of the new state.

Sound files are loaded into the AudioManager in StateHandler to avoid the need for loading each time the state is changed. Audio is then accessable by all State classes through the AudioManager pointer that is passed to them. The sound for the motor is enabled and disabled in GameState’s Init and Release functions. The motor audio’s pitch is altered every frame by a Map function that maps the pitch to the speed of the motor. The motor sound is of a dirt bike idling. As the pitch increases, it increases the frequency and gives the sound effect of the bike accelerating.

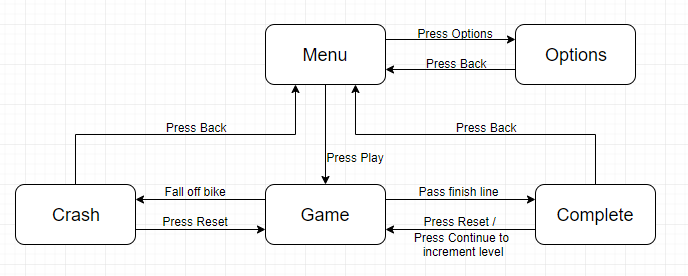
State stores a pointer to StateHandler and an enumerator indicating the type of state it is. It has pure virtual functions, HandleInput, Update, Render and Release.

All child classes must override State’s pure virtual functions with their own implementations. As they inherit from State, they all have access to State’s member variables and functions. They all have their own functionality on top of that. They have their own variables and functions that are specific to them.

When StateHandler creates instances of these different child classes, it creates them all as States.

This is called polymorphism. It allows for child classes to be stored as objects of their parent class’ type. This allows StateHandler’s state pointer to take on the form of whatever child class of State is needed.

Having this state handler means that all the state changes can be handled from inside the current state so all of the input handling is encapsulated nicely.

State Machine Diagram:

The diagram above shows how the states interact with each other and how the user can navigate the states.

MenuState is what the user is greeted by when loading the game. It is a simple state. It makes use of the SpriteRenderer class to display some graphics. It shows the title and the controls for the game and gives the user the ability to select between playing the game and the options screen. It has a GenerateNewTarget function that uses the LERP (Linear intERPellation) function to move the rider sprite in a pseudo random pattern around its origin to simulate it flying through the air.

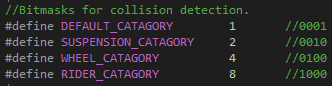
OptionsState sets the level variable in Statehandler that is then passed down to the State classes. This happens when the user leaves the options screen to go back to the main menu.

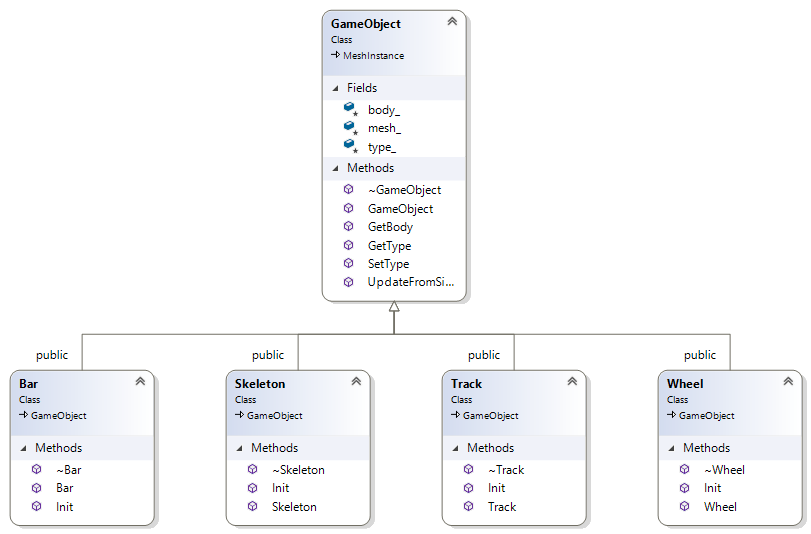
The CrashState and WinState give simple options to navigate back to the main menu or restart the level. WinState has a continue option that goes back to GameState but also increments the level.

GameState is relatively complex and has multiple objects associated with it. The b2World is initialised here and meshes are loaded. Bike is an object that belongs to the GameState which has multiple instances of GameObjects.

GameObject contains an enumerator for the type of object it is, a pointer to a box2D Body object and a pointer to a GEF Mesh object. Mesh is the 3D model associated with the game object. The box2D Body is the shape of the hitbox in the 2D plane.

It has a generic UpdateFromSimulation function that uses matrix transformations to orient the mesh to match the body’s location and rotation.

The header file for GameObject defines some integers that are used as bitmasks for use with collision detection. This is a box2D feature that uses the concept of category bits and mask bits. Category bits are the bits associated with the body. Mask bits are the bits the category bits are compared with in a collision of bodies.

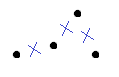
Each subclass of GameObject has unique Category bits. This ultimately allows for selecting what an object can collide with.

Skeleton, Wheel, Track and Bar are all child classes of GameObject. They share GameObject’s attributes. The only difference is what parameters are passed into their Init functions. Depending on these parameters, different shaped bodies are generated and different bit masks are used to determine what the object can interact with.

When the game is closed, the CleanUp function in SceneApp is called that handles deleting any leftover objects on the heap, to avoid memory leaks and calls StateHandler’s Release function that unloads audio files from the AudioManager safely.

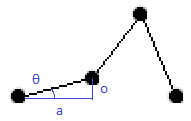
Techniques Used

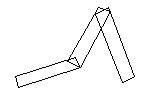
Vectors are declared for the position of each of the joints of the bike.



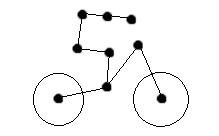
From these joint positions, the centre points between the joints are calculated. These are the origins of the bodies making up the bike. The wheel origins are at the joints.



The orientation of the bodies were calculated using the atan2 function in the standard library. The parameters passed into this were the difference in x and y positions for each pair of joints.

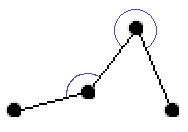


The length of each body was calculated for the collision box. This was calculated using Pythagoras theorem. Each of the bodies of the bike has a simple rectangular hitbox.

The rider is made up in a similar way. The length of the rider’s limbs are declared. As are the starting angles of the joints.

The position of the rider’s ankle joint is hardcoded to a point close to the bike’s back suspension joint. The rest of the joint’s positions are calculated by originating at the previous joint’s position, calculating the angle of all the previous joints combined and projecting out at this angle, the distance of the length of that limb. The body’s origins and angles are calculated in the same way as the bike.

When creating the bodies, they are passed their origin, orientation and length. They are also passed a density variable that can be tweaked when simulating the physics.

The angles between each of the bodies was calculated by simply subtracting the angles of each of the bodies. These were used in the creation of box2D joints.

Revolute joints are used for all of the joints except for the front suspension which is a Prismatic joint and the neck which is a weld joint.

There is a motor on the back wheel that has a set amount of torque and the speed is adjusted by the user.

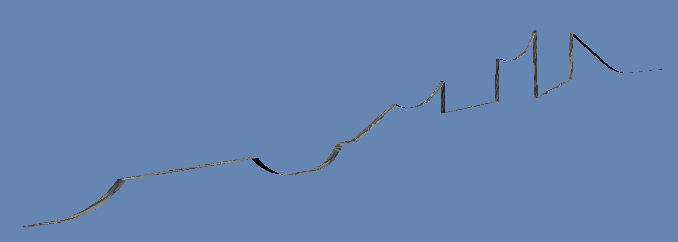
There is a motor on the front wheel that is turned on and off with a speed of zero to simulate a brake.

There is a motor on the back suspension joint that is always trying to rotate to its upper limit, to simulate a rotational spring.

The ankle, knee and hip joints also have motors that are turned on and off to simulate muscles and to achieve the rider leaning forwards and backwards. The arm joints do not have motors as it was unnecessary. They do however have limits to simulate the human body (the limbs cannot invert). The prismatic joint uses an equation to simulate spring type behaviour.

When joints are applied, they force the bodies into position. It was important to calculate all of the positions and angles of the bodies however as if this wasn’t done the joints would be under a lot of “stress” and can cause undesired behaviour with forces acting against each other. The results can be quite comical with the bike folding in on itself or exploding.





The level is created using Blender. There is a self-made algorithm that reads the object file and extracts the vertices from it to be used in the box2D chain shape. The pseudocode for the algorithm is as follows:

Load the object file into a string.

Loop through every character in the string.

If the character is a ‘v’ then it is the start of the vertex data.

Continue to the beginning of the number.

Save the position in the string.

Continue to the end of the number.

Store these characters in a string.

Convert the string to a float.

Continue to the beginning of the second number.

Repeat the process.

Store these values as a point.

Add the point to an array if it is not a duplicate of one already there.

Sort the points into order by x value from lowest to highest.

This allows the developer to edit levels very quickly. This was very important as a lot of tweaking was involved to get the ramps the right size and the right distance apart to make the level enjoyable. The track is a made up of planes with a normal facing in each direction. This design was chosen mainly because of the big frame rate increase that was achieved as opposed to having it as a solid object.

More liberal use of assets could have been achieved without the framerate being such an issue if the program was designed with more of a focus on data locality.

The bike’s mesh is created by the developer from scratch, however the design was inspired from “Trials – Frontier”. The rider is adapted a lot from a free model, found online. A lot of effort was put in learning blender for this project.

Data Oriented Design

Object Oriented Programming is a style of programming that structures the code in such a way that code is split up nicely into encapsulated chunks. Objects can inherit from other objects that have common properties and can be treated as instances of these objects. Code is logically structured using abstraction and it is easy to add and remove objects as well as re-use them. It is easily maintainable. Objects can be stored in arrays and can be iterated over, calling functions on each individually. This seems intuitive but can also have drawbacks.

One of the pitfalls of Object Oriented Programming is the fact that objects’ member variables are going to be broadly spaced out in memory. When looping over these objects, there will be a lot of jumping around in memory. This makes bad use of the CPU’s cache. Trips from the CPU Central Processing Unit) to RAM (Random Access Memory) along the bus are expensive.

When the CPU retrieves data from RAM, it does not just grab data at that specific memory address. It also retrieves data that is stored around it. How much data that is retrieved depends on the size of the CPU’s cache. Accessing memory on the cache is a lot faster than accessing memory in RAM. If the data the CPU needs to fetch is stored on the cache, this is called a cache hit. If the data is not on the cache, this is a cache miss and the CPU will have to go and fetch another chunk of memory from RAM. While this is happening the CPU has to wait for the data and will be idle until it is retrieved. This is going to dramatically increase processing time. If the data the CPU is trying to access is located in random locations in RAM the probability for cache misses is high.

The idea of Data Oriented Design is to make as much use of the cache as possible. If the data you are needing to access is laid out contiguously in memory then when data is retrieved from RAM, it will also retrieve data that is likely to be needed in the subsequent clock cycles.

If you can identify where cache misses are happening, you can try to optimise your code here. It is important to identify “hot” and “cold” data. “Hot” data is data that will be accessed a lot during runtime. An example of this could be data that is accessed during the game’s update loop, where it will be accessed every frame. “Cold” data is data that will only be accessed rarely. For example, a specific interaction that only happens once during a level. “Hot” data benefits the most from Data Oriented Design. With “cold” data, the benefit is much more limited.

One practice that can lead to bad use of the cache is over use of pointers. This will result in having to jump around in memory more, which will lead to more cache misses. In object oriented programming there is often a lot of passing pointers to objects down the hierarchy of classes.

An example of bad use of the cache in this game is in GameState’s Update function. GameState’s Bike object has many box2d joints. These are declared on the heap. Little control is had over where the memory manager will store these so they will likely all be scattered in memory. Every frame, every box2d joint in Bike is accessed to be compared in some way. The CPU is pointer chasing to find the data. The cache is going to have to be constantly renewed as there will be a lot of cache misses. This is called cache thrashing. A way around this would have been to declare these objects on the stack in an array. This would lead to more cache hits and less CPU down time.

Also, some of the variables that are being accessed from Bike, like joint angles, which are floats, are declared next to GameObjects. GameObjects are large in comparison to floats and they are not needed in the subsequent code. These will be taking up a lot of room on the cache so there will not be as much room for relevant data. More thought into how these are laid out could increase performance.

If you have a class where there are large objects that are not used much, it can be worth creating a pointer to the object and declaring it on the heap so that only the pointer is pulled onto the cache to make room for more relevant data. Although you will most likely get a cache miss when you need to use this data, the performance gain of having more space on the cache for relevant data could well outweigh the cost of occasionally retrieving this specific data through a pointer.

Some drawbacks of Data Oriented design is you lose a lot of flexibility in the way you structure your code. You cannot make use of polymorphism and inheritance in the same way as you want to structure your data in a contiguous fashion. You lose abstraction (the ability to manage complexity by obscuring irrelevant details) which will potentially make the code harder to read.

User Guide

When the user first loads the game, they should be greeted with a menu screen with a motorcycle and the title of the game. From here, the user can navigate the menu with the UP button and the DOWN button. To select an option from the menu, the user can press the CROSS button.

In the Options Menu, you can navigate the level selection with the LEFT button and the RIGHT button. You can press the SQUARE button to go back to the Main Menu. Whichever level is selected when the user leaves the options menu will be the level the user goes into when they play the game.

In game, the user can accelerate with the CROSS button, brake with the SQUARE button and lean backwards and forwards with the LEFT button and the RIGHT button respectively. To reset the level, the user can press the TRIANGLE button at any point.

The best technique for gaining speed is keeping the back wheel connected to the ground as much as possible and trying to land nicely on the landing ramps to have the force of gravity on the user’s side. If the jumps are over shot, they can lose speed as a result and not have enough speed to make the next jump.

If the rider makes contact with the ground, the rider will be disconnect from the bike. His limbs will become floppy and he will fall for three seconds before the user is greeted by a splash screen, asking them if they would like to continue or return to the main menu.

There is a cruelly placed bar at the end of the levels that acts as a finish line. The user will need to pass this bar to complete the level. (Only the bike needs to get passed it!)

Conclusion

It was very important for this type of game that the simulation felt believable and fun to play. The way the bike and rider were implemented, with the ability to make them any shape, made the process of tweaking positions and angles a lot less painful. The ability to tweak these variables was invaluable. However this did take a lot of time to implement correctly. There is actually some external software called RUBE (Really Useful box2d Editor) that has a GUI (Graphical User Interface) that allows the user to select objects representing box2d bodies and join them together with “joints”. The user can change variables to do with the joints and can see a simulation of it. It then generates the code to paste into the user’s code. This method does seem like cheating but would have saved a lot of development time.

One thing that could be done to tidy up the code in the bike class would be to use structs to hold variables of the same type, as there is a long list of variables. This would help to reduce the lines of code and help make it more readable.

One problem encountered when creating my state machine was that each state needed to include the header file “state\_handler.h” but StateHandler also needed to include “state.h”. This is called circular dependency and this will not compile. To tackle this problem, forward declarations were used. The “state.h” header file now forward declares the StateHandler class. This is possible as there are only pointers to the class in the header file, it is never initialised or used here. The “state\_handler.h” header file is then included in the associated cpp file. This is a solution to a common problem that can be applied to future projects.

Thinking about different ways of structuring code and balancing between efficiency and readability have been very thought provoking and will definitely be taken into account in the future. It has been eye opening, learning about how physics engines work and how matrix calculations are used in real applications. Gaining some 3D modelling skills and how to apply them has been interesting. A lot of confidence has been gained in the ability to create 3D, physics based games.

Experience has been gained working within a third party framework. A skill that will be very useful in industry. Working on this project has highlighted the importance of Data Oriented design and when to possibly apply these methods in favour of an object oriented approach.

References

Nystrom, R. (2019). *Data Locality · Optimization Patterns · Game Programming Patterns*. [online] Gameprogrammingpatterns.com. Available at: http://gameprogrammingpatterns.com/data-locality.html [Accessed 8 May 2019].

Motorcycle Helmet V2 3D Model – Available at – <https://free3d.com/3d-model/motorcycle-helmet-v2--886241.html>

Bogie Mountain Bike 3D Model – Available at - <https://free3d.com/3d-model/mountain-bike-bogie-768642.html>

Rocks Sculpted Textures – Available at - <https://www.unrealengine.com/marketplace/en-US/slug/rocks-sculpted-textures>