SOFT 323 Documentation

Student: Tim Garner

Student ID: 10234237

Table of Contents

Software Versions

Visual Studio Version

The visual studio used to produce the project was “Visual Studio 2010 Service Pack 1”. This is available on all University computers, and also via the MSDNAA website to students free of charge.

DirectX SDK Version

DirectX 10 has been used to produce this project. It was released in November 2006 and is exclusive to Windows Vista and newer operating systems. It consisted of extensive rewrites to the code base, depreciating many of the functions available in DirectX 9. Most noticeably, the fixed-function pipeline is no longer supported, requiring the use of (potentially) more confusing shaders (Microsoft, 2012)

End User’s Guide

From the End User’s perspective the Tiger ‘fly’s’ like an aeroplane. You can apply a “thrust” which will push the Tiger in the direction it is currently pointing. Releasing the thrust will cause the Tiger to slow down, and also be overcome by the effects of gravity, gradually being pulled to the floor. The Tiger can also drop bombs, which will bounce on the floor in a realistically believably way before coming to a resting state.

Controls

Keyboard

- W: Thrust forwards
- Left Arrow: Yaw left (Turn left)
- Right Arrow: Yaw right (Turn right)
- Up Arrow: Pitch up
- Down Arrow: Pitch down
- Space: Drop bomb

XBOX 360 Controller

- Left Joystick: Thrust forwards
- Right Joystick, Left: Yaw left (Turn left)
- Right Joystick Right: Yaw right (Turn right)
- Right Joystick, Up: Pitch up
- Right Joystick Down: Pitch down
- Right Shoulder Button: Drop bomb

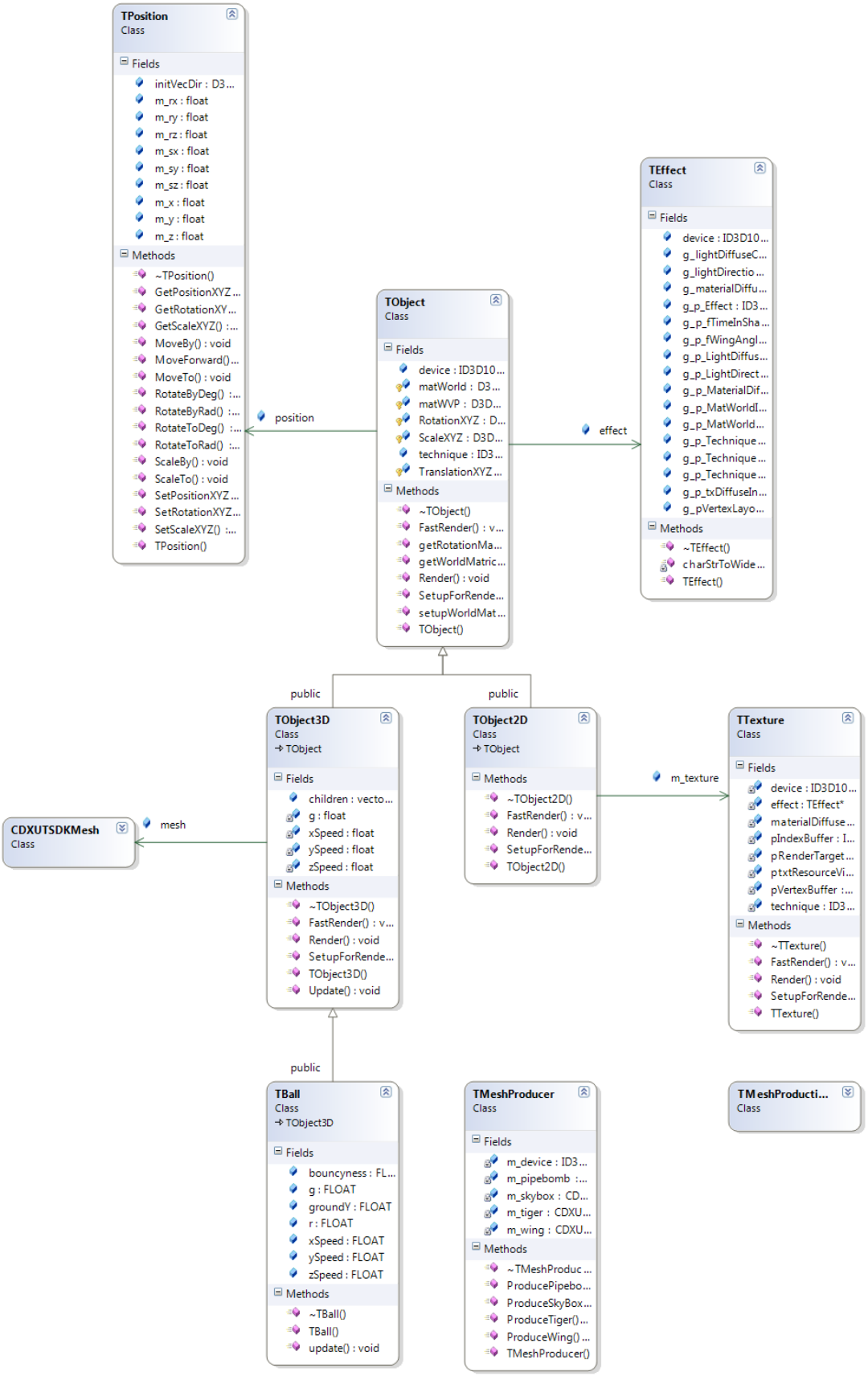
Bombs can only be created twice per second, and will drop ‘metronomic-ally’ if space is held. A sound will be played when they are dropped letting the player know they have released their cargo.

You cannot fly lower than the ground level, even when outside the ‘floating platform’ made up of floor tiles. The Tiger is ‘trapped’ within the Skybox. In reality this means the skybox is always centred on the location of the Tiger, this stops the tiger from ever reaching the edge. The pitch can only reach a max/min value of 35 degrees.

Programmer’s Guide

To give a quick representation of the Programs structure, a Class Diagram has been produced:

Figure 1 Class Diagram



Classes produced by myself are prefixed with the letter T. A structure has been created that allows common features shared between 2D and 3D objects to be shared. The base class TObject ‘has-a’ TPosition and a TEffect.

TPosition is concerned with the objects position, scale, rotation, and translation within 3D space. It provides a set of convenience methods to manipulate these properties safely.

TEffect holds all the information regarding the shader file for that object. It is a pointer type; so many objects can all share the same effect object, this helps reduce code clutter, and can help with performance.

A TObject3D descends from TObject, sharing all public and protected properties, and importantly, implements the pure virtual methods, SetupForRender, FastRender and Render. These are pieces of code that setup the shader with the correct variables, and calls the Render function of the mesh. The mesh is a pointer to a CDXUTSDKMesh, many TObject3D’s can share the same mesh, this increases performance dramatically when rendering, and also uses much less RAM as only one copy of the mesh is needed.

TObject2D is similar to TObject3D, except it implements the virtual methods in a 2D appropriate way. Each TObject2D has a pointer to a TTexture; this stores the actual image file and acts in a similar way to the CDXUTSDKMesh in that the same texture can be shared between multiple 2D objects. The benefits of this can be seen when creating a floor based on hundreds, even thousands of tiles. Loading the program without shared textures can cause delays of 10’s of seconds, however with shared textures it loads up with no noticeable delay.

A TBall implements physics on a TObject3D, allowing the ‘force’ of gravity to act on the ball.

Additional convenience classes are TMeshProducer and TMeshProductionLimiter. TMeshProducer is used when creating a TObject3D, you ask the mesh producer for a pointer to the mesh. The mesh producer will deal with the details of deciding whether to create a new object, or just return a pointer to an existing one. This allows developers to reuse the same mesh hundreds of times without worrying about negative performance.

TMeshProductionLimiter is used to rate limit the production of meshes, this is useful for example when dropping bombs, as the user is only allowed to create a bomb every half second.

Evaluation

I am pleased with the structure of my program, specifically the modulatory I have achieved by refactoring different sample code and adding my own to create a base I feel satisfied to move forward with for the second piece of coursework.

The physics appear to be quite believable, and I am happy with how the program ‘feels’ when you play it. However I do think the physics concerning the flight of the Tiger could be improved. I have also forgone spending more time on fancy effects, in trade off I have spent more time in improving the health of the code base in general.

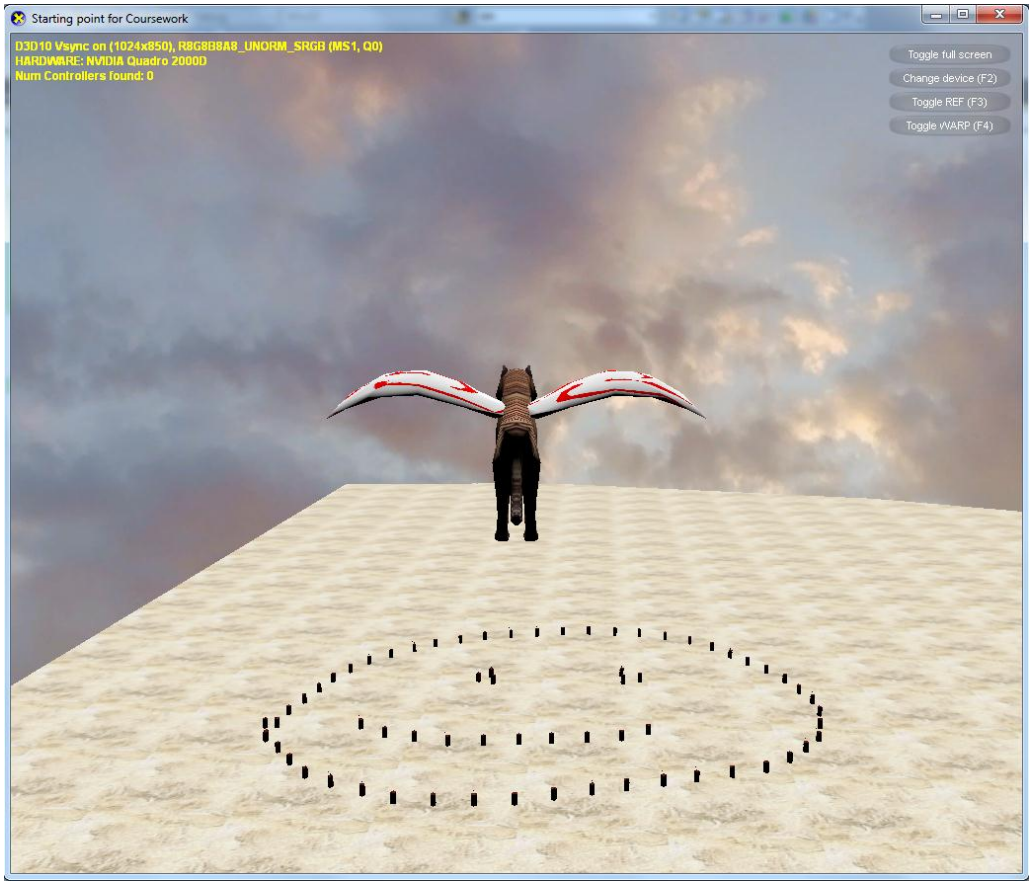
However, I am particularly proud of the realistic flapping of the Tigers ‘wings’. This is produced by a custom vertex shader that manipulates the wing mesh. It is based on the cosine of the current frame time, plus the distance away from the tiger body. This is then multiplied again by the distance away from the tiger body to create a new value of y for the particular vertex. This creates a realistic flap motion as the effect is more pronounced as the momentum travels down the wing.

Another notable feature is that a parent TObject3D keeps track of any associated children. Once the parent has rendered itself, it then renders all of its children, passing down its world matrix as a ‘context’.

I have also spent some time focusing on performance gains; this is manifested by having three render methods, SetupForRender, FastRender and Render. If used correctly they can dramatically improve the frame rate when rendering many of the same objects. SetupForRender will, as the name suggests setup all of the correct variables in the shader for that object type. This is attributes like the vertex layout and the primitive type. FastRender will then just calculate the world matrix, and matrix view projection for that instance of the object before finally rendering it. Render can be used to chain both steps together.

All other base, and physics objectives of the coursework have been implemented, and I am happy with my personal understanding of how each part works together. I have also spent a considerable chunk of time using the PIX for Windows DirectX profiler/debugger to track down and remove all memory leaks from the program.

I will end with a screenshot from my game:



References

Microsoft. (2012, 11 28). *Deprecated Features (Direct3D 10) (Windows).* Retrieved 01 14, 2013, from Microsoft.com: http://msdn.microsoft.com/en-gb/library/windows/desktop/cc308047(v=vs.85).aspx