Developer Journal

Serge Radinovich

23-07-2015: *Working on Concept Document, GDD and TDD*

We have been working on detailing the game's theme, core mechanics and technical requirements. Before we settled on the side-scrolling competitive arena game, we had discussed a slower-paced, cooperative strategy game with a top-down view. This idea involved randomly generating tiles that the players would navigate. These tiles would intermittently drop off, so the players would have to strategize over their moves. Ultimately we felt that this game idea would not have a compelling 30-second gameplay loop. While it would have some interesting technical requirements, we were very keen to make a core loop that was really engaging (while also being technically complex).

Because I had developed a DirectX 10 framework already, it was agreed that I would focus on implementing our core Win32 application and DirectX10 rendering components. We therefore set up a product backlog outlining the basic requirements of the application and renderer so that I could get under way.

30-07-2015: *Basic Win32 Application and DirectX 10 Rendering Framework*

Today I worked on creating a game engine framework that would form the basis of all the systems of the game. The main requirement was to enable textured 2D quad rendering through DirectX 10. I had an existing framework to work with which employed a number of advanced 3D rendering techniques so I focused on simplifying the API and limiting it to a small number of functions so that the others could pick it up easily and change it as they wish. The following is a summary of my implementation.

In order to run the rendering framework, the user must grab a pointer to the singleton AppWin32 object. This object has an initialization function which takes in the handle to the instance of the application and a reference to a GameRoot object. This GameRoot object is an implementation of the IRoot interface which houses all of the framework's game-critical sub-systems such as the renderer. The IRoot provides the AppWin32 with an interface to execute frames from within the win32 message pump. Initializing the IRoot causes the DX10Renderer singleton to be initialized and this object provides all the rendering functionality required to draw textured quads. The general frame loop therefore goes from the AppWin32 message pump, to the IRoot update function and then to the corresponding sub-system update functions (for example the renderer's update).

The DX10Renderer provides an interface to the GameRoot and its game-critical objects (which inherit from IActor) through static functions. So, for example, a Player object could initialize itself by calling DX10Renderer::initializeRenderTask2D(…). This function is a centralized initializer which handles loading of required shaders and corresponding vertex input layouts for the IActor. This is a very simple and rigid API at the moment which only allows for one way of assigning shaders and textures to IActors. Each IActor contains a RenderTask2D object which houses all of its rendering details after it has been initialized by the DX10Renderer. The mesh pointer and pointers to effects file variables such as textures are kept here. At the moment there are many unnecessary variables in the RenderTask2D class which are artefacts from the 3D system this framework was derived from.

In its current state, the framework can successfully render textured quads with alpha.

31-07-2015: *Integrating Box2D into the Framework*

Now that a basic renderer has been established, it was time to integrate a 2D physics system. I had worked with Bullet Physics in my 3D framework and so working with Box2D came naturally. It is a relatively simple, light-weight and clean API and I did not want to obscure it from the other team members by overuse of abstraction mechanisms. Also, we are not entirely sure about our needs for the API (for example, we have not decided if we need any collision shape apart from a square). Therefore, my implementation is small and simple. I focused on integrating the API into the framework in a way that was consistent with the renderer (as the rendering and physics systems are both sub-systems of the engine). The following is a summary of my implementation.

The IRoot now contains a pointer to the PhysicsSystem singleton object in the same way it does regarding the DX10Renderer. As such, the PhysicsSystem is initialized and updated in the same manner as the renderer. Just as the IActor derived clases use static functions from the DX10Renderer class to initialize their rendering, IActors use the PhysicsSystem class to initialize their rigid bodies and collision shapes. The PhysicsSystem provides two static functions which correspond to the creation of the rigid body and a fixture between a collision shape and that rigid body, respectively.

The current implementation allows users to instantiate 2D box collision shapes and rigid bodies so that the rendered quads can move around and collide based on the physics system. I have also ingrained a static perimeter in the game space through the PhysicsSystem initialization function so that objects will not leave the screen space.

03-08-2015: *Adding Keyboard and Xbox Controller Input*

We had all implemented DirectInput (Keyboard and Mouse) and XInput (Xbox Controller) systems in earlier assignments so adding input was never going to be a big ordeal. My implementation is as follows.

The IRoot contains an InputHandler object which allows the GameRoot to call functions in order to detect game-specific events. The InputHandler is a singleton which houses both the XInputManager and the D3DInputManager. This allows the input handler to detect input with all devices through single function calls. For example, handleJump() can detect a push of the space-bar on the keyboard or the A-button on the controller.

The next step in relation to input is to figure out how to allow for local multiplayer with the Xbox Controllers.

11-08-2015: *Animating sprites*

Right now we have a system whereby each game actor has a single transform and texture. I needed to work on this to allow for each actor to hold an array of textures that could be animated during the game loop. This means that actor initialization needed a substantive rework so that developers could provide the API with a vector of strings relating to texture files for the respective actor. My current implementation iterates textures on the CPU, rather than on the GPU through a shader. This is acceptable because we will not need to optimize the game to achieve a stable framerate, given it is a simple 2D game. For the sake of testing, I animated two textures constantly on the player sprite. The two textures were dependant on the player's current state so that if he was jumping, they would be different to the two textures used when standing on a surface.

18-08-2015*: GDD work*

We have been working on finalizing the Game Design Document together for a while now. All of the work has been allocated to individuals. When the GDD is completed we can formulate our Product Backlog. I have already been feeding basic components of the game framework into the Product and Sprint Backlog so that I could set everything up for the team. This way we can hit the ground running because we will have a solid foundation upon which we can cooperatively develop the game with.

We are still discussing some mechanics critical to gameplay. For instance, Austin is keen on implementing 3 different melee attacks whereas I would like to see a single melee attack and a projectile attack (which could be reflected by the melee attack). Austin's reservations about my implementation are that it is too derivative of Samurai Gun.

***24-08-2015**: Pre-Production Phase wrap up*

This week we are focusing on completing the GDD, TDD framework, Product Backlog, Pre-Alpha Spring Backlog and playable prototype. We are also aiming to create a presentation in which we can show-case some of the art assets and game-flow diagrams.