Kelvin Lam 15051161

Best Engine Documentation

The Hague University | 3D-engine design

2018

# The engine

For the KB-01 course of The Hague University we got the assignment to design a 3D-engine. By the end of this course the engine must have multiple distinct features, or requirements, that must be working before the deadline. These requirements include:

* A Skybox
* Assets loaded in once
* Moving/rotating entities
* Terrain loaded with a height map
* A bitmap loader
* Camera movement
* An abstract renderer
* A scene/level loader
* No memory leaks
* Support for multiple windows
* Input of mouse/keyboard
* Debug logger

The engine was made with C++ in Visual Studio. We have a demo project called ExecutableProject, but if you want to link it yourself to your own project that’s fine too. The engine itself is called GameEngine3D, bit of a generic name, our group name is Best Engine now but I can’t really be bothered to change the name now.

# General Classes

**AbstractAsset**with this interface you can create your own asset to use in the AssetManager.

**AssetManager**This class makes it so that we are able to use our and your created assets in the engine. It creates and loads the assets into our engine, the engine then does the processing and releasing on its own, however you can do that yourself if you’d like that better.

**Bitmap**A bitmap is an image file with structured data for reading the file itself. We use our Bitmap class to chart the pixels of our bitmap file into an array. We make sure the bitmap file had to have a bit count of 24 so the data is read correctly.

**Camera**Camera is a default entity we’ve created, we put in logic for a free roaming camera, but you can create your own camera from Entity, you can use the SetViewTransform method of the passed renderer to change the view.

**Debug**With the Debug class we are able to log error messages to text files. Inside our logger we can document things that happen within our engine, such as information of what ErrorType it is (ERRORTYPE\_INFO, ERRORTYPE\_WARNING or ERRORTYPE\_ERROR), which file and line it happens in and the message that was passed to it.

**Entity**With this interface you can create your own entities to load into the Scene with SceneManager and link a loaded asset by setting m\_assetName to the name of the loaded asset. You can also set m\_entityType with an EntityType (ENTITYTYPE\_NOTYPE, ENTITYTYPE\_SKYBOX, ENTITYTYPE\_TERRAIN, ENTITYTYPE\_MODEL or ENTITYTYPE\_CAMERA), which changes the way a Scene will update the entity (with ENTITYTYPE\_CAMERA it gets a target position the position of the camera which a Skybox can use, ENTITYTYPE\_SKYBOX will set the position of the skybox to the position of the target camera position). You can also override Update(float delta) and use that as if it’s a game loop.

**FPSCounter**With this class we can get the FPS (Frames Per Seconds), at which the engine is running and the delta time. Using the Timer header file, we can count the frames that have passed within a certain amount of time. This number is eventually returned to us with via GetFPS().

**AbstractInput**With this interface you can create your own input handler.

**Kernel**The Kernel is one of the most important parts in the engine. Form this class we set the given renderer and input and use them inside the game loop. We also loop through all the windows and update and process the active scene.

**Model**With the Model class we load, render and release meshes with the default DirectX 9 pipeline through the given renderer. It inherits from AbstractAsset.

**AbstractRenderer**With this interface you can create your own renderer and use it in our engine.

**Scene**In the Scene class we add to, get and remove Entities from a scene.

**SceneLoader**With SceneLoader you can load assets, create scenes and add entities to those scenes with an XML file. You can learn how to create the XML file in the generated documentation (Refer to Installation and References).

**SceneManager**With this class we add, get and remove scenes, you can use this to access a specific or all scenes.

**Skybox**With the Skybox class we create a skybox with a cubemap, then load the generated vertices to the renderer. It inherits from AbstractAsset.

**Terrain**With the Terrain class we create a terrain from a heightmap and a texture, then load the generated vertices to the renderer. It inherits from AbstractAsset.

**Timer**The Timer file is for keeping track of the time within the engine. It is mostly used for keeping track of the FPS of the engine.

**Vector3**We can create an x, y and z of any passed value (Vector3<float> for example, there are typedefs for Vector3i (int) and Vector3f (float) however).

**Window**In window we create a window and save its position, size, title handle and instance.

**WindowManager**You can’t see the engine work without a window to look through. This is where the WindowManager class comes into play. In this class we have a list of Window(s) and handle the creation of a Window.

# DirectX 9 Specific Classes

**Direct3D**The renderer we created for DirectX 9, it creates the Direct3D device and processes all the loaded assets in the current active scene and releases them when necessary, you can also get the created device with GetDevice() to use in DirectX 9 specific classes.

**DirectInput**With our input file we can monitor when certain keys are pressed and we move our mouse button. We use these inputs for our camera so we can move around in the world our 3D-engine creates. When input isn’t working, we can send an error message to our Debug logger so we can see where things go wrong.

# Demo project

At the beginning of the main function we start with the Kernel. With the Kernel we make sure that a window is opened where we can display relevant entities on screen. After the Kernel is initialized you can create another window if you so wish to.

You can create your own entities by inheriting from Entity, we have only created one entity ourselves which is a camera, but you don’t have to use our camera, you can also just create your own camera and load it into the scene with SceneManager.

You can create Assets by inheriting from AbstractAsset and use them in AssetManager wherever you want in your program, you can link an entity and an asset by using the name of the asset you created for the m\_assetName you inherit from Entity.

With the SceneManager you can add, get, process although we do that for you, and remove scenes and add entities to the scene you want.

With SceneLoader you can load assets, scenes and entities into the scenes with an XML file. A more detailed explanation is in the generated documents (Refer to Installation and References).

And the near-final but very important piece is to enter the game loop with the EnterLoop(); function from Kernel. After that you can release and remove assets since that would be the end of the game. But all entities, assets and scenes automatically release and remove after that point.

**The smallest setup possible of the engine**  
 se::Kernel kernel("WindowName", true, 0, 0, 800, 500, new se::Direct3D(), new se::DirectInput());

kernel.EnterLoop();  
  
And the values are of course replaceable, your own title, your own size, your own renderer and your own input.

# Installation and References

You can get to the installation guide with links to relevant webpages and the generated code documentation in Documentation (aka current folder of this file) -> html -> index.html, open index.html with a web browser and you’ll see the installation guide pop up or use the shortcut in the Documentation folder.