Kelvin Lam 15051161  
Eimantas … ---  
Niels van der Meer 15112411

KB-01 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
* Moving/rotating assets
* A height map
* A bitmap
* Camera movement

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.

**AbstractCamera**With this interface you can create your own camera to load into the CameraController. It is connected to the Entity header file, with that connection we’ve defined movement for our camera inside our engine using both the rotation of our camera and movement, you can do the same yourself.

**CameraController**The CameraController is used to put your (or our) defined camera into the game loop, you can also transform the camera after the fact.

**Debug**Inside our Debug files we are able to create our debug logger. Inside our logger we can document things that happen within our engine, such as information about what is happening and warning or error messages when that is needed. An example of this is getting a message when input is failing.

**Entity**Our Entity works from our Transform header file. The transform file can give us the position of an entity in the engine, like the camera. 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 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().

**Input**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.

**Kernel**The Kernel is one of the most important parts in the engine. Form this class we manage our camera control, the opening of our windows and the rendering of everything within out engine. Also from this file we can send error messages to our logger if something isn’t working the way it should or an instance is missing. The loop that keeps on rendering every frame also is put in motion within this file.

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

**Scene**In Scene we add into, and get and remove entities from a scene. We also set, update and render the terrain and skybox since both are usually only loaded once.

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

**Skybox**With this interface you can create your own skybox to load in the Scene with SceneManager.

**Terrain**With this interface you can create your own terrain to load In the Scene with SceneManager.

**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.

**Transform3f**We can get the X, Y and Z of position, but also the rotation and the scale. With those values, you can work with all the transformations inside the engine.

**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 define how big the window is we create and are also able to define to be able to shut down the window, we can also create and remove additional windows if necessary.

# 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.

**DirectXCamera**The camera we created for DirectX 9, it sets all the relevant transformations (position, rotation) and passes it to the Direct3D device.

**DirectXMesh**The mesh creator we created for DirectX 9, it creates the mesh from an .x file, loads the materials and textures on it, and processes it and can also release it.

**DirectXSkybox**The skybox creator we created for DirectX 9, it creates the skybox with the given transformation and puts the skybox texture on it with the given source. It also processes it and releases it when it needs to.

**DirectXTerrain**The terrain creator we created for DirectX 9, it creates the terrain with a given grayscale heightmap and loads textures over it with the given texture source. It also processes it and releases it when necessary.

# 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 set the cameracontroller if you have created your own camera. Or if you wish to load our camera which is freeroam with input on our WASD keys for actual movement in the faced direction and use input on the arrow keys for the rotation of the camera.

You can create Assets 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 and set a created Skybox and/or Terrain to the scene you want.

A skybox is created with a Transform3f and a texture source, the skybox always follows the camera. We create our terrain with a given grayscale bitmap file for height and a texture source to put on the created terrain. For any entity you can set the position, rotation and the scale.

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.

# 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.