Division 3D Game Engine Manual

Authors: Nick Dekker, Marthe Veldhuis
Engine by: Nick Dekker, Marthe Veldhuis, Tobi van Westerop
GitHub: https://github.com/martheveldhuis/DivisionEngine

Table of contents

1. Introduction

In the simplest way, the Division 3d game engine can be used to display a scene from a scene file on a window. This will only take 3 lines of actual code. For setting up the engine, refer to section 2, this is necessary for every use of the engine. To use the engine in the simplest way, please go to section 3. To learn more about scene files and how they are constructed, refer to section 4. On the other hand, it is possible to manually set up the engine, create a scene, add custom windows, add entities, etc. This requires a more experienced user that wants to customize their own experience. The manual for this sandbox interface can be found in section 5. Examples for both these uses of the engine can be found in the “DivisionGame.cpp” file.

Enjoy the Division experience!

2. Project setup

For setup, link the .lib file in your game project and include the engine's kernel which contains the engine's interface methods. The engine's components are all located inside the "Division" namespace, so make sure to use the scope resolution operator preceded by "Division" to access the engine. Note that for the current implementation, DirectX 9 and Windows will be utilized.

```
#include "Kernel.h"
```

You can also simply use the provided "DivisionGame" project in which this is already set up, and customize it to your needs. A unit test project is also included called "DivisionTest" which could be used when expanding the engine. Currently only 2 classes are covered.

The project could be run without having to open it in an IDE and accessing the code by running the appropriate executable for your system (64-/32-bit) found in the "Executable" folder.

3. Simplest engine use

Initialize the engine

To make use of the Division 3d game engine, you must initialize it in your game project. For this purpose you must instantiate the kernel of the engine. By default, the DirectX 3D framework will be initialized.

```
Division::Kernel divisionEngine;
```

Load a scene

Next you must specify a scene file which contains information about what is inside the scene (see section 4 for more information about scene files). The scene will be built based on the information in this file. If this file is left empty, defaults will be set for you, which creates a scene with only a skybox. The scene specified below loads a scene with a skybox, texture and one entity.

```
divisionEngine.loadScene("example_scene.json");
```

Run the engine

The last step is to run the engine so that the scene will be displayed.

```
divisionEngine.run();
```

4. Scene file format

The scene files that are loaded using the load scene method have to adhere to a certain format. A scene file is a .JSON file which contains specific components which will form a scene when loaded in the engine. Make sure the scene file is saved in the “DivisionGame” folder, so the engine can load it properly.

An example scene file is included within the engine called “example_scene.json”. Note that each element has at least a “name” component. This name must be unique to differentiate the objects in the code. The different elements of this example will be run through.

4.1 Scene

The first element is the scene object. It consists of a name, a terrain, and a skybox texture. If no scene object is written in the file, a scene will still be created with all defaults.

Name is the name of the scene, so how the scene will be referenced throughout the code. If no name is filled in, it will be defaulted to “scene1”.

Terrain consists of two sub-elements, a heightmap file and a texture file. These files make up the terrain. The heightmap file must be a .bmp file or it will not be parsed correctly. The texture can be any of these formats: .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm, or .tga. Both these elements are required to create a terrain. If these are not set, they will be defaulted to the heightmap and texture files filled in below. If you want to create a scene without a terrain, leave out the terrain component completely and no defaults will be set.

The skybox texture is the image that makes up the skybox around the player. If this object is not written in the scene, it will be defaulted to the file filled in below. For this texture file, the same file formats are allowed as by the terrain’s texture.

```
"scene":{
    "name": "scene1",
    "terrain": {
      "heightmap": "terrainhm.bmp",
      "texture": "terrain.bmp"
    },
    "skybox_texture": "skybox.bmp"
  },
```

4.2 Renderers

The next items are the renderers. Since multiple renderers can be included in a scene, make sure to use the square brackets to indicate an array.

The only field that is available for these is the name of how the renderer is referenced in the code. In the future, this could be expanded to include the type of renderer (e.g. DirectX/OpenGl) and the behaviour. Again, if no renderers are included in the file, it will default to the one below.

```
"renderers":[
  {
    "name": "DXrenderer"
  }
],
```

4.3 Entities

Entities are objects that are rendered in the scene. These are always optional, so if you wanted a scene without entities you could remove this from your scene file. The example scene has two entities. An entity has many sub-elements which can be seen below.

Again, the name is how the entity will be referenced in the code. The mesh is the .x file of the entity containing the structure of that entity, and the texture is the texture you want to render over that mesh. The texture can be any of the file format mentioned before in the scene. If any of these are not filled in, they will be defaulted to the values seen below.

The pos and angle features are both optional and will be defaulted to 0 if omitted. Each have an x, y and z component. If you choose to set the x, y or z values, you have to fill all three of them or the scene file will not be read correctly.

```
"entities": [
  {
    "name": "tiger1",
    "mesh": "tiger.x",
    "texture": "banana.bmp",
    "pos": {
      "x": 6,
      "y": 0,
      "z": 10
    },
    "angle": {
      "x": 0,
      "y": 0,
      "z": 0
    }
  },
  {
    "name": "tiger2",
    "mesh": "tiger.x",
    "pos": {
      "x": 7,
      "y": 0,
      "z": 0
    },
    "angle": {
      "x": -1,
      "y": -1,
      "z": 0
    }
  }
],
```

4.4 Windows

Windows are essential to the rendering of the scene, so if no windows are present in the scene file, it will be defaulted to the values below. The sub-elements will also be filled in with these values if they are not included.

The name property represents the name of the window in the code, while the window title is the name that is shown in the title bar of the rendered screen.

The renderer field specifies with which renderer that window will be rendered. This should reference the name of a renderer specified before in the scene file. If this is not the case, a default renderer will be created.

```
  "windows":[
    {
      "name": "window1",
      "window_title": "First window",
      "renderer": "DXrenderer"
    }
  ]
}
```

5. Sandbox interface use

In the sandbox interface, the main file will contain every step of using the engine. This means you have to set up a scene, the entities in it, and run the engine.

The first step of this type of use is different from the simplified use, as you can specify the type of repository that is created. However, since only the DirectX repository is currently implemented, the effect is the same.

```
Division::Kernel divisionEngine(Division::REPOSITORYTYPE_D3D9);
```

After creating the kernel object, it can be used to retrieve the manager classes, as well as the framework repository. The manager classes may be used to create a scene and entities for that scene. The following functions will provide you with the managers:

```
Division::ResourceManager* resourceManager = divisionEngine.getResourceManager();
Division::SceneManager* sceneManager = divisionEngine.getSceneManager();
Division::Repository* repository = divisionEngine.getRepository();
```

Next, a scene can be created. This is done using the scene manager. In the code example below, we can see a scene being created, using the internal reference “scene1”.

```
Division::Scene* scene = sceneManager->createScene("scene1");
```

This scene is now empty, but can be filled using entities, a terrain and a skybox. However, we first need a window with an attached renderer and camera to show these objects in. Here, a window is created, and placed at the location 0,0 on the screen with a dimension of 720x720. The title of the window will be “First window”. Multiple windows can be added to one scene.

```
Division::Window* window = repository->getWindow("First window");
window->moveWindow(0, 0, true);
window->resizeWindow(720, 720, true);
```

A renderer should be created to be in charge of rendering the objects on the screen. First, we create it. The user should call the ‘setup’ function on the renderer to set the default rendering options. In future releases of the engine we plan to expand this method to be able to pass arguments about the options of the renderer. For instance to set the viewport, or render state options to different values from the defaults. Different options can currently only be set by getting the device from the renderer, and setting the (DirectX) render states directly on the device. Lastly, the renderer should be added to the scene manager.

```
Division::Renderer* renderer = repository->getRenderer();
renderer->setup();
sceneManager->addRenderer("renderer", renderer);
```

A camera object should also be created, to create a viewpoint. Since the camera is an entity that could have textures and meshes it needs a reference to the resource manager. Also, a x, y and z position can be passed to put the camera on a different position in the scene than 0, 0, 0. Currently, the camera entity is not yet rendered.

```
        Division::Entity* camera = repository->getCamera(resourceManager, 5, 5, 5);
```

Now that we have a window, a camera and a renderer, they should be added to the scene. We can do this by using the “addWindow” function on the scene. They will be linked to each other by the scene. The first string represents the internal reference to the window in this case it is called “window2”. This way, the window can always be retrieved to be deleted.

```
        scene->addWindow("window2", window, renderer, camera);
```

Lastly we can add entities, the skybox and a terrain. A mesh needs to be set to determine the shape of that entity, using a .x file. The default entity textures (when they exist in the .x file of the mesh) can be overridden by calling the “setTexture” method as shown below.

```
        Division::Entity* tiger = new Division::Entity(resourceManager);
        tiger->setMesh("tiger.x");

        Division::Entity* skyBox = repository->getSkyBox(resourceManager);
        skyBox->setTexture("skybox.bmp");

        Division::Entity* terrain = repository->getTerrain("terrainhm.bmp",
resourceManager, "terrain.bmp");
```

After the entities are created, they will have to be added to a scene to be rendered. The skybox should not be added as an entity, but bet set as the scene’s sky box with the “setSkyBox” method.

```
        scene->addEntity("terrain", terrain);
        scene->addEntity("tiger1", tiger);
        scene->setSkyBox(skyBox);
```

The scene has now been filled and the engine can be run to enter the game loop.

```
        divisionEngine.run();
```