EE3K1 Interactive 3D Design for Virtual Environments and Serious Games

Specification and Workplan

Yousef Amar (1095307)

Introduction and Design Considerations

Oil spills can be the result of several different occurrences such as damage to underwater pipes, accidents on offshore oil rigs, and leaks on oil tankers amongst many others. One such event is the Exxon Valdez oil spill in 1989, an oil tanker that may have released up to 119,000m^3 of crude oil into the ocean (Bluemink; Anchorage Daily News, 2010).

For this project, main focus will be on educating players, specifically on the effects and repercussions of tanker oil spills on the caliber of the Exxon Valdez oil spill, mainly on wildlife. The simulation will endeavor to immerse the player in an environment where a myriad of animals are dying or suffering. The player will be required to navigate the scene of the oil spill and rescue as many animals as possible within a certain time boundary. The addition of this game mechanic, in conjunction with convincing 3D design, aims to evoke deep-rooted emotions in the player that hopefully have a long standing effect on them.

The player can move around in a boat to survey the scene or to interact with the environment. On the surface, the damaged tanker on fire as well as the extensive oils slick can be seen, not to mention the animals of course. Through pressing a key, the player will also be able to leave the boat and toggle an underwater mode with a similar objective, only to later re-enter the boat on collision and pressing the relevant key.

Implementation Plan

Geometry

Given the nature of this project, there exists a heavy emphasis on models. Thus far, a skydome, ocean, and oil tanker are in place (see screenshots in appendix). In addition to these, there will be several models that would either have to be downloaded online or created from scratch with polygon count and suitability for real-time rendering. These include:

- Birds – These may require simple rigging for a desperate flapping animation
- Marine Life (such as starfish, dolphins and other simple fish pending research)
- Potentially people to set the scene such as firefighters or drowning victims
- A small boat, preferably with a polygonal mesh rather than Bezier curve surfaces

Textures, Materials and Lighting

It is especially important that the models used are realistic enough to reach the goals of this simulation. For living creatures, animation alone would not be enough. For instance, the textures of the birds and marine fauna may have to be edited to include an apparently overlaid layer of oil and the materials darkened. It would be a good idea to increase specular lighting on animals that have been “oiled” for a wetter, waxier visual feel.

For global sunlight, a simple hemi light source is used. This, together with reddish point light sources on fires and spot lights for the player's flashlight, should prove adequate enough to convey the atmosphere. To give underwater navigation a more realistic feel, a fragment shader will be used to darken all pixels and make them bluer as the player descends deeper and deeper. With some simple math, this darkening can be made to affect the outer edges of the viewport more than the middle to give a softer "tunnel-vision" view.

In order to not nuke the FPS count, the water is a simple, static plane with two textures; an animated normal map for lighting the wave effect, and another texture for reflection. No actual objects will be reflected but it is impossible to tell as the illusion of tiny waves and ripples makes it seem like the sky is being mostly reflected and all other light scattered (see appendix).

AI/Game Logic

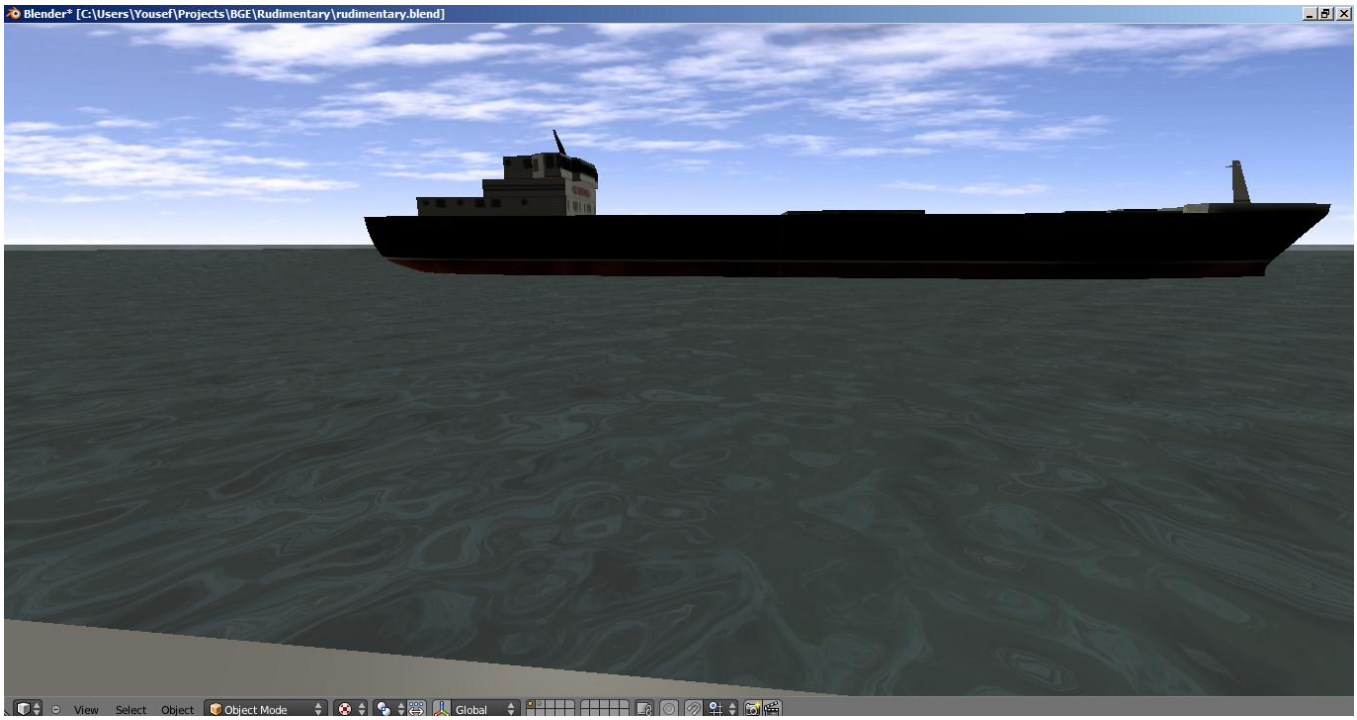
So far, the player can move around in a first person view using simple keyboard sensors and motion actuators. Additionally, a simple python script to handle mouse motion and allow for a mouse-look mechanism has been implemented with the aid of the slightly outdated but still relevant BGE documentation (see appendix). Horizontal mouse motion rotates the actual player while vertical rotates a child game object called "Head" on its local x-axis so that the player remains upright. This is done twice; once for the player's scene and another time in the skydome scene to rotate the skydome which is rendered as a background scene to remove any notion of depth from itself.

Fortunately all real-time physics is already handled by the included bullet physics engine and all that would be left to do is create random paths for the animals using a random sensors and motion actuators (similar to the maze example on the WebCT tutorials).

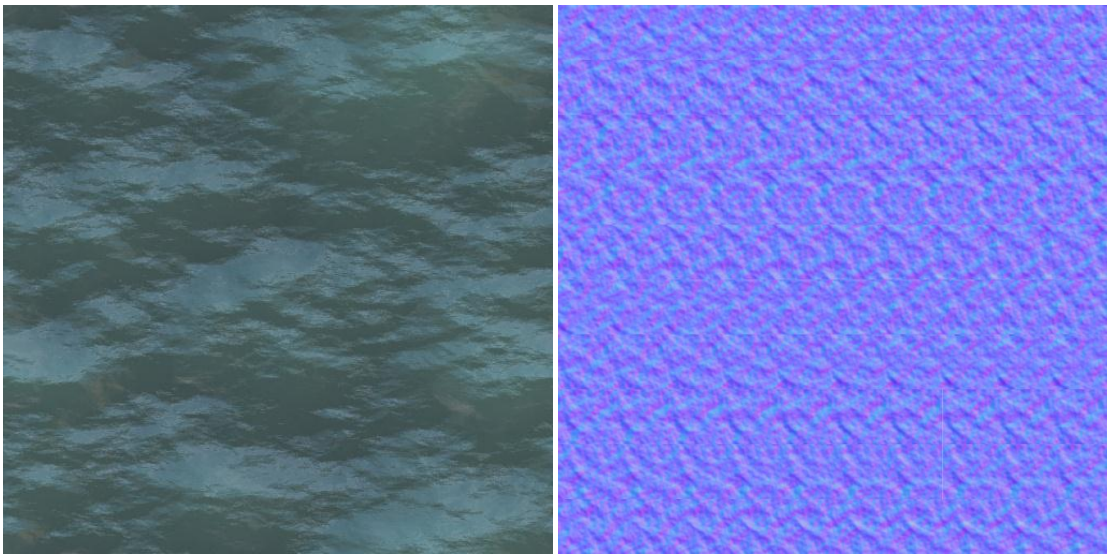
Smoke and fire would be simulated through a makeshift particle system as blender particle systems do not work in the game engine. For example, smoke would need an object to act as an emitter, with an always sensor that triggers an "Edit Object" actuator in "Add Object" mode. It would spawn smoke particles, stored in a different layer, at a set frequency that grow and rotate slightly and move upwards with random variations. This would not require key frame animation as the same effect can be achieved with just game logic. Furthermore, individual particles would not be affected by lighting, shadows or physics. The purpose of this is to build atmosphere.

The scoring system would encompass a simple integer game property to store the score that is incremented every time the player collides with one of the animals, thus "saving" them, while at the same time removing the object from the scene. This should allow for a reasonable tradeoff between realism and performance but can always be built upon if need be. Through the extra entertainment, hopefully the player would stay engaged more to further reinforce the goal of education through empathy. This should allow for a more memorable experience as opposed to a plain simulation or simply fixing an underwater pipe.

Appendix



Player view of intact oil tanker, ocean and sky



Reflection and normal map frames, courtesy of Sam Dearn (YouTube tutorials)

Player logic; notice the "Head" child object that includes the mouse-look script

```
import bge
import Rasterizer

def main():
    cont = bge.logic.getCurrentController()

    # Get mouse movement sensor
    mouseMove = cont.sensors['MouseMove']
    # Get camera actuator
    actCam = cont.actuators['Camera']

    # Get viewport dimensions
    width = Rasterizer.getWindowWidth()
    height = Rasterizer.getWindowHeight()

    # Get mouse position relative to viewport centre
    x = int(width/2) - mouseMove.position[0]
    y = int(height/2) - mouseMove.position[1]

    # Initialize mouse to prevent jerk the first time (technique found online)
    if not cont.owner['mouseInit']:
        # Set boolean mouseInit property to true
        cont.owner['mouseInit'] = True
        x = 0
        y = 0

    # Apply rotation
    cont.owner.applyRotation((0.0, 0.0, x*-0.004)) #Skydome
    actCam.object.applyRotation((y*0.004, 0.0, 0.0)) #Camera

    #Already done in Main scene
    #if x != width/2 or y != height/2:
        # Center mouse in game window
        #Rasterizer.setMousePosition(int(width/2), int(height/2))

main()
```

Skydome mouse-look script (included as the player mouse-look was significantly more complex)