Project 0: Level Design

Due Date: Friday, February 10th, 11:59 PM

1 Introduction

The purpose of this assignment is to familiarize you with our game engine Dojo and get you thinking about what makes a game fun to play. As such, in this assignment you will be creating a map using GtkRadiant (or the Quake 3 map editor of your choice) and incorporating it into Dojo. Because Dojo incorporates physics (via the Open Dynamics Engine), we envision a marble navigation game, such as Marble Blast by Garage Games. You will load your level into Dojo and deal with issues such as character control, basic game interaction, triggers, etc... While the technical side of this assignment is very important, remember that the purpose of games is to entertain. Creativity and fun are necessary parts of this assignment; if you don't think your map is interesting, we probably won't either.

2 Requirements

For this assignment you must create a map that is playable using our game engine Dojo. Your map must incorporate triggers (for things such as scoring points, teleporting, extra lives etc.) that your character can interact with. We provide a few example triggers but you should also create your own. We realize this assignment probably seems pretty open ended, but that's how we designed it. Beyond these few requirements, your grade for this assignment will be based primarily on how creative you get with your map and triggers.

3 Creating a Map

To create a map you will use GtkRadiant (or whatever Quake 3 map editor you want as long as it works with our support code). GtkRadiant is a free cross-platform map editor available at http://www.qeradiant.com. **Make look at the GtkRadiant help session slides if you are not already familiar with a map editor**. For this assignment you will have to create a map and create a series of triggers that affect game play (see 5).

4 Dojo

After creating your map and exporting it to .bsp format, you will run it in Dojo. **Make sure you look at the Dojo help session slides**. To use your map with Dojo you must first copy your bsp file to **(your dojo directory)**/scratch/build/install/q3maps/maps and your textures to **(your dojo**

directory)/scratch/build/install/q3maps/textures. You can then edit the line in App.cpp that reads:

Q3Map* map = Q3Map::create("Map", "q3maps/", "example.bsp");

to read:

Q3Map* map = Q3Map::create("Map", "q3maps/", "<**my map name**>.bsp");

You can now run Dojo and it will load your map.

4.1 Loading a Map

Dojo already has code to load Quake 3 maps (see BSPMAP::Map, Dojo::Q3Map and Dojo::Q3MapModel) so all you have to do is use it. If, however, you find some cool feature in the Quake 3 file format that Dojo does not support feel free to augment the map loading code however you like. In fact, if you do something really cool and let us know about it you'll probably get bonus points!

4.2 Heads Up Display (HUD)

The HUD package provides basic functionality to display icons, buttons, text, or fullscreen backgrounds in Dojo. Use the GameUI class to control the HUD. There are two ways to create the HUD: one, you can call all the GameUI::add* functions in your C++ code (see Demo::onInit and Demo::onGraphics), or (two) you can pass it a Lua script file that allows the HUD to build itself. Here are a few examples:

File: hud.lua

```
assert(G_UI)
x = 50
y = 400
width = 72
height = 216
AddSprite(G_UI,"gamesprites/sidebar.bmp",x,y,width,height)

text = {font = "font/venusrising.fnt",
                size = 14,
x = 50,
y = 60,
color = {1, 0.5, 0.5 }, -- optional
outline = { 0, 0, 0.5, 1 }, -- optional
text = "hello world"
}
```

```
AddText(G_UI,text)
AddButton(G_UI,
        50,80,25,25,"gamesprites/neutral.png",
"gamesprites/hover.png",
"gamesprites/down.png")
```

The nice thing about using the Lua script is that you can change the look of your user interface without recompiling your code, which you would have to do if you use the C++ functions.

Note that all of the Lua functions take a parameter, G_UI. This is very important. It will be explained in the Lua helpsession.

To display the HUD, call GameUI::doUIGraphics and pass it a RenderDevice. In order to achieve realtime effects like mouseover button changes, call GameUI::update with the current mouse position. The RealTime parameter is there to support animating UI items, which don't exist yet. To see if a button was clicked on a mouseup event (because the button doesn't actually 'release' until a mouseup), call GameUI::doClick. It takes the position of the click, and passes back a boolean (true if a button reacts) and a reference to the button's handle. You can then act accordingly (quit if the quit button was pressed, etc).

In order to manipulate the items, use the GameUI::set* functions, or call functions on the UIItems themselves.

4.2.1 Character Control

We have provided you with basic character controls (Demo::onUserInput), it is up to you whether or not to make any changes. We have also provided simple mouse based camera controls (Demo::onSimulation), again, feel free to change them however you wish.

5 Triggers

Part of your assignment is to create triggers in your map. Triggers can be things such as jump pads, telelporters, free lives, etc... GtkRadiant has a series of built in triggers called entities. To place an entity in GtkRadiant right-click on the grid view of your map and a menu with a series of entities will pop-up. Once you place an entity you can move it around the map using the standard GtkRadiant controls. Some entities, however, must be attached to an object already within the map in which case you must first select that object and then place the entity. There are also entities that need to be targeted (connected) to another entity (usually a **target_position**). To do this, place the **target_position**, then select the entity to connect it to and press ctrl+K. If this is done correctly an arrow should appear linking the two. Entities can have different parameters so consult the GtkRadiant documentation (and go to the GtkRadiant help session). You may want to have a trigger that teleports the player or respawns the player etc., and to do this you will want to manually change the player's position

using Entity::setFrame. However, manually changing the player's position while ODE is processing physics wil cause an error. As a result, you must find some way to call Entity::setFrame outside of the physics callbacks (i.e. outside of Entity::onCollision).

Dojo loads entities into a list BSPMAP::BSPEntity objects that you can access using Q3Map::entityArray(). We have given you examples of how to use this information, see TriggerFactory.[*h*|*cpp*],Trigger.[*h*|*cpp*]. Each BSPMAP::BSPEntity contains the following information:

```
Vector3 position - the location of the entity

std::string name - the name of the entity

int spawnflags - the value of the spawflags option

std::string target - the entity this points to (the empty string
if no such entity exists)

std::string targetName - can be used the same way as target, but
depends on the specific entity (be sure to look at the GtkRadiant
documentation)

int modelNum - the index into the list of BSPModels[1] that this
entity points to (-1 if it points to nothing)

std::string otherInfo - any other info loaded from the bsp file
(check the documentation for each specific entity)
```

For the most part you can interpret these however you want, but there are a few exceptions.

- **info_player_deathmatch** - this is used to specify the starting location of your character.
- **light** - this is used to specify lights in the map.
- **target_position** - this is a special entity used to target entities such as trigger_push or trigger_teleport. See TargetedPushTrigger.[*h*|*cpp*] for an example of how to use this.
- **path_corner** - you can use this if you want, but it will be used in the next assignment so you may want to avoid using it to avoid confusion.

It is your responsibility to create the functionality of the triggers in Dojo. We have provided code to create a TriggerFactory for you in App.cpp and examples of how to write triggers:

For examples of different types of entities see ScoreTrigger and TargetedPushTrigger.

[1] A BSPMAP::BSPModel is a specific type of geometry in the bsp file format. The main information that you need to consider for this assignment are the max and min values that each BSPModel stores, which are the max and min points of the bounding box around the geometry. The list of BSPModels can be accessed using Q3Map::modelArray()

6 Getting Started

After attending the help sessions, begin by running Dojo with the example maps we provide to get a feel for how the character controls work, and to see the example triggers. The more familiar you are with the game mechanics, the better you will be able to design your level. We have provided you with a sample map (dojo/scratch/build/install/q3maps/maps/example.bsp) to get you started.

7 Handing In

7.1 What to Hand In

Make sure that you hand in all of your code as well as your map and textures. Along with this, you must handin a README in which you explain your design choices in creating your map as well as the changes you made to Dojo.

7.2 Handin Script

IMPORTANT: Make sure to clean your solution (Build>Clean Soluction in Visual Studio) before handing in.

To handin your project, run /u/course/cs196-2/bin/cs196-2_handin in your project's the base directory.

Appendix: Where Things Happen in Dojo

Loading a Map: BSPMAP.cpp, BSPMAP.h, BSPMAPLoad.cpp

Loading your Map: App.cpp

Creating Triggers: TriggerFactory.cpp, TriggerFactory.h

Main Rendering: Demo.cpp, World.cpp

Character and Camera Controls: Demo.cpp

UI: GameUI.cpp, GameUI.h

Interfacing with ODE (physics): Entity.cpp, Entity.h, World.cpp, World.h