Liam Dick

S1218100

Games Programming 3

Coursework Documentation

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.*

*Liam Dick*

**Contents**

Code Explanation 3

* Main 3
* Player 11

References 13

Class Diagram 14

Story Board 15

**Code Explanation**

This document will cover the most relevant parts of the code within the project, this will cover the codes purpose and functionality.

**Main.cpp**

The managers

First point of entry into the after the declaring the windows attributes is the declaration of the various managers which are vital to the project, the first being the window manager:

static cWNDManager\* pgmWNDMgr = cWNDManager::getInstance();

Following this is the addition of the input manager which is responsible for getting the inputs from devices such as keyboard and mouse:

static cInputMgr\* theInputMgr = cInputMgr::getInstance();

This is attached once the managers have been declared in :

pgmWNDMgr->attachInputMgr(theInputMgr);

The font manager is responsible for displaying text on the window overlay, this uses fonts provided by the user, this will be covered later.

static cFontMgr\* theFontMgr = cFontMgr::getInstance();

The sound manager’s responsibility is providing the scene with various sounds which can be themes that provide background music in a loop or quick sound effect sounds:

static cSoundMgr\* theSoundMgr = cSoundMgr::getInstance();

The Spheres

To create the planetary bodies for the star system I will have to create spheres using the sphere class (“cSphere.cpp). Using this I create three spheres:

cSphere theSun(4, 40, 40);

cSphere thePlanet(3, 30, 30);

cSphere theMoon(1, 20, 20);

It’s here that the sphere objects are assigned their names and are given variables that affect their radius, the number of slices created and the number of stacks used.

The Textures

The next step is to retrieve the textures that are going to be used for the models which are later going to be loaded. The textures will have to be loaded from the Model asset folder and assigned their meaningful identifiers:

cTexture spaceShipTexture;

spaceShipTexture.createTexture("Models/SpaceFighter/mat\_ship.png");

cTexture spaceGateTexture;

spaceGateTexture.createTexture("Models/SpaceGate/mat\_gate.png");

cTexture spaceStationTexture;

spaceStationTexture.createTexture("Models/SpaceStation/mat\_stat.png");

cTexture spaceSatelliteTexture;

spaceSatelliteTexture.createTexture("Models/SpaceSatellite/mat\_sate.png");

cTexture laserTexture;

laserTexture.createTexture("Models/laser.tga");

Then other textures are loaded which are going to be used for other rendered objects like the spheres created earlier which will be textured to create the planets. Another texture is required to create the star field, this star field will be discussed later.

cTexture starTexture;

starTexture.createTexture("Images/star.png");

cTexture planetTexture;

planetTexture.createTexture("Images/ice.png");

cTexture sunTexture;

sunTexture.createTexture("Images/Sun.png");

cTexture moonTexture;

moonTexture.createTexture("Images/Moon.png");

The Star field

cStarfield theStarField(starTexture.getTexture(), glm::vec3(50.0f, 50.0f, 50.0f));

The star field is the rendering of one texture which is multiplied into a very large sphere around a determined point in the scene, this is an effective way on creating a skybox type background.

The star system

This part will cover the creation and initialisation of the miniature solar system in the scene.

Firstly the spheres and their respective names are initialised along with their respective position in the scene:

theSun.initialise(sunTexture.getTexture(), glm::vec3(0, 0, 40), glm::vec3(0, 0, 0));

thePlanet.initialise(planetTexture.getTexture(), glm::vec3(10, 0, 20), glm::vec3(0, 0, 0));

theMoon.initialise(moonTexture.getTexture(), glm::vec3(10, 5, 8), glm::vec3(0, 0, 0));

This creates the sun in the middle of the scene and for it to act like the heart of the game (as well as centre reference point for the player). The planet is created a short distance away from the sun with a moon in orbit of the planet. To create a more realistic solar system the planets and moon should rotate slightly, these values are provided by:

float planetRotSpeed = 3.0f;

GLfloat planetOrbit = 0.0f;

float moonRotSpeed = 5.0f;

GLfloat moonOrbit = 0.0f;

Light

The sun provides the scene a light source which is an obvious necessity. This done by declaring instances of the light class and assigning their relevant variables to control the light generated.

cLight sunLight(GL\_LIGHT0, lightColour4(0, 0, 0, 1), lightColour4(1, 1, 1, 1), lightColour4(1, 1, 1, 1), glm::vec4(0, 0, 20, 1),

glm::vec3(0.0, 0.0, 1.0), 0.0f, 180.0f, 1.0f, 0.0f, 0.0f);

cLight lfLight(GL\_LIGHT1, lightColour4(0, 0, 0, 1), lightColour4(1, 1, 1, 1), lightColour4(1, 1, 1, 1), glm::vec4(30, 0, 100, 1),

glm::vec3(0.0, 0.0, 1.0), 0.0f, 180.0f, 1.0f, 0.0f, 0.0f);

cLight rfLight(GL\_LIGHT2, lightColour4(0, 0, 0, 1), lightColour4(1, 1, 1, 1), lightColour4(1, 1, 1, 1), glm::vec4(-30, 0, 100, 1),

glm::vec3(0.0, 0.0, 1.0), 0.0f, 180.0f, 1.0f, 0.0f, 0.0f);

cLight cbLight(GL\_LIGHT3, lightColour4(0, 0, 0, 1), lightColour4(1, 1, 1, 1), lightColour4(1, 1, 1, 1), glm::vec4(0, 0, -100, 1),

glm::vec3(0.0, 0.0, 1.0), 0.0f, 180.0f, 1.0f, 0.0f, 0.0f);

Fonts and Sound

The font locations are declared which then are added into an array to store them and to assist in managing them:

LPCSTR gameFonts[3] = { "Fonts/digital-7.ttf", "Fonts/space age.ttf", "Fonts/doctor\_who.ttf" };

theFontMgr->addFont("SevenSeg", gameFonts[0], 24);

theFontMgr->addFont("Space", gameFonts[1], 12);

theFontMgr->addFont("DrWho", gameFonts[2], 48);

In addition to assigning the fonts to array each respective font is assigned a meaningful name, retrieves the correct position of the array created earlier and given a font size before being assigned to the font manager.

Sound operates in a similar manner by retrieving the sounds from their asset folder and assigning them to an array, such sounds include explosions, gun shots and the main music theme:

LPCSTR gameSounds[4] = { "Audio/Breaking Ground.wav", "Audio/shot007.wav", "Audio/explosion2.wav", "Audio/engine\_2.wav" };

theSoundMgr->add("Theme", gameSounds[0]);

theSoundMgr->add("Shot", gameSounds[1]);

theSoundMgr->add("Explosion", gameSounds[2]);

theSoundMgr->add("Engine", gameSounds[3]);

Models

In this section the model loading process is covered. The first step is to create instances of the model loader class and assign names to the models that will be loaded, additionally each model will require its .obj file and the texture assigned earlier, as seen in:

cModelLoader spaceShipMdl;

spaceShipMdl.loadModel("Models/SpaceFighter/spaceship01.obj",spaceShipTexture);

cModelLoader spaceGateMdl;

spaceGateMdl.loadModel("Models/SpaceGate/gate.obj", spaceGateTexture);

cModelLoader spaceGateMdl1;

spaceGateMdl1.loadModel("Models/SpaceGate/gate.obj", spaceGateTexture);

cModelLoader spaceGateMdl2;

spaceGateMdl2.loadModel("Models/SpaceGate/gate.obj", spaceGateTexture);

cModelLoader spaceStation;

spaceStation.loadModel("Models/SpaceStation/station.obj", spaceStationTexture);

cModelLoader spaceSatellite;

spaceSatellite.loadModel("Models/SpaceSatellite/satellite.obj", spaceSatelliteTexture);

cModelLoader theLaser;

theLaser.loadModel("Models/laser.obj", laserTexture);

After this it is time to create the player controlled model. This starts with the call to the Player class (cPlayer.cpp) which then initialises variables like position, rotation, scale, speed etc.:

cPlayer thePlayer;

thePlayer.initialise(glm::vec3(1, 0, 70), 90.0f, glm::vec3(0.002, 0.002, 0.002), glm::vec3(0, 0,0), 1.0f, true);

To fully create the player object more is required to allow control from the inputs captured by the input manager for the various key presses and the sound manager to create any relevant sounds that are generated by the player’s actions

thePlayer.attachInputMgr(theInputMgr);

thePlayer.attachSoundMgr(theSoundMgr);

The next step is adding the other models which are classified as enemies. The enemies are instances of the enemy class (cEnemy.cpp) and contain information that assigns the other models to these instances. The other models involve the space station, satellite and the space gates.

cEnemy gate;

gate.initialise(glm::vec3(1, 0, 60), 0.0f, glm::vec3(0.02, 0.02, 0.02), glm::vec3(0, 0, 0), 1.0f, true);

gate.setMdlDimensions(spaceGateMdl.getModelDimensions());

cEnemy gate1;

gate1.initialise(glm::vec3(10, 0, 60), 0.0f, glm::vec3(0.02, 0.02, 0.02), glm::vec3(0, 0, 0), 1.0f, true);

gate1.setMdlDimensions(spaceGateMdl1.getModelDimensions());

cEnemy gate2;

gate2.initialise(glm::vec3(20, 0, 60), 0.0f, glm::vec3(0.02, 0.02, 0.02), glm::vec3(0, 0, 0), 1.0f, true);

gate2.setMdlDimensions(spaceGateMdl2.getModelDimensions());

cEnemy station;

station.initialise(glm::vec3(1, 0, 50), 0.0f, glm::vec3(0.02, 0.02, 0.02), glm::vec3(0, 0, 0), 1.0f, true);

station.setMdlDimensions(spaceStation.getModelDimensions());

cEnemy satellite;

satellite.initialise(glm::vec3(10, 0, 50), 0.0f, glm::vec3(0.002, 0.002, 0.002), glm::vec3(0, 0, 0), 1.0f, true);

satellite.setMdlDimensions(spaceSatellite.getModelDimensions());

It should be noted that the scale of these objects have been greatly reduced several decimal points, this is due to the models being massive naturally and was required to be scaled down. Each of these “Enemies” have their own variables in terms of position and rotation much like the players initialisation.

The camera

Again similar to light to be able to see the objects it would be desirable to have a camera to move around and actually see the objects without relying on the default debug view. I decided to go for a third person camera which is done by assigning the players position and lookAt values which is seen here:

cCamera theCamera;

theCamera.setTheCameraPos(glm::vec3(thePlayer.getPosition().x, thePlayer.getPosition().y, 3.0f + thePlayer.getPosition().z));

theCamera.setTheCameraLookAt(glm::vec3(thePlayer.getPosition().x, thePlayer.getPosition().y ,0.0f));

This allows the camera to follow just behind the player and is constantly updated each time the player moves. I decided to add in a cameraSwitch variable which will allow the player to switch from a third person camera and a fixed perspective view. This works by assigning two keys which when pressed will switch between the two respective views. This is controlled in the main class with a couple of if statements:

if (thePlayer.cameraSwitch == true)

{

outputMsg = to\_string(thePlayer.cameraSwitch);

theCamera.setTheCameraPos(glm::vec3(10.0f, 3.0f, 70.0f));

theCamera.setTheCameraLookAt(glm::vec3(0.0f, 0.0f, 0.0f));

theCamera.update();

}

if (thePlayer.cameraSwitch == false)

{

outputMsg = to\_string(thePlayer.cameraSwitch);

theCamera.setTheCameraPos(glm::vec3(thePlayer.getPosition().x, thePlayer.getPosition().y, 3.0f + thePlayer.getPosition().z));

theCamera.setTheCameraLookAt(glm::vec3(thePlayer.getPosition().x, thePlayer.getPosition().y, 0.0f));

theCamera.update();

}

This changes the cameras view depending on the key pressed which allows simple and reliable interchanging of view within the scene.

In a similar fashion it was decided that the sound for the main theme should be able to be toggled between an on and off state if the player prefers. This again will change the value of a Boolean variable which will stop and start the audio, this is seen here:

if (theInputMgr->isKeyDown(69) && thePlayer.soundToggle == false)

{

theSoundMgr->getSnd("Theme")->playAudio(AL\_LOOPING);

}

if (theInputMgr->isKeyDown(82) && thePlayer.soundToggle == true)

{

theSoundMgr->getSnd("Theme")->stopAudio();

}

Rendering

Rendering then takes place after the window is created and the sound and camera functions are declared.

The starfield discussed earlier is rendered followed by the solar system (sun, planet and moon). The sun has its light values activated as well as its material being applied:

theSun.prepare(0.0f);

sunMaterial.useMaterial();

sunLight.lightOn();

lfLight.lightOn();

rfLight.lightOn();

cbLight.lightOn();

theSun.render(theSun.getRotAngle());

In a similar fashion the other spheres are created with the exception of a lack of a light but the inclusion of a rotation variable which allows the planet and moon to gradually rotate.

Following this the models are called to render and each with their retrieval of attributes of position, rotation and scale that were declared earlier.

The laser model is an exception as it is generated in an iterator that creates multiple instances of the laser model each time the player presses the fire laser key.

The next part of note is the generation of the text overlay on the screen. This is performed by retrieving the required font that was assigned a name earlier and then prints the desired text on screen with a position and colour which is shown here:

theFontMgr->getFont("Space")->printText("E & R = Sound ON/OFF", FTPoint(830, 45, 0.0f), colour3f(255.0f, 255.0f, 255.0f));

Fog

An additional mention for the main class is the addition of fog. This is carried out by calling GL functions like GL\_FOG. The first step was the declaration of variables which will be used by the fog:

GLuint filter;

GLuint fogMode[] = { GL\_EXP, GL\_EXP2, GL\_LINEAR };

GLuint fogfilter = 0;

GLfloat fogcolour[4] = { 0.5f, 0.5f, 0.5f, 1.0f };

The filter and colour variables are simple variables which will be used by the function however fogModel variables GL\_EXP and GL\_EXP2 are equations which are components on generating the fog, the equations are as follows:

GL\_EXP

F = e - (density.c)

GL\_EXP2

F = e - (density.c) 2

The GL\_Linear equation is:

This is then applied to the fog rendering code:

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

glFogi(GL\_FOG\_MODE, fogMode[fogfilter]);

glFogfv(GL\_FOG\_COLOR, fogcolour);

glFogf(GL\_FOG\_DENSITY, 0.05f);

glFogf(GL\_FOG\_START, 1.0f);

glFogf(GL\_FOG\_END, 5.0f);

glEnable(GL\_FOG);

Here the fog model and filter is applied along with the fog colour which is declared (I decided on a simple grey mist colour). Following this the density is set which is another factor into the earlier equations of GL\_EXP and GL\_EXP2. Then the start and end is declared which governs the Linear equation.

**cPlayer.cpp**

Explanation

The main purpose of the player class is the union with the input manager class to allow the user to control and manipulate the player controlled model.

if (m\_InputMgr->isKeyDown(VK\_RIGHT))

{

rotationAngle += 1.0f;

}

if (m\_InputMgr->isKeyDown(VK\_LEFT))

{

rotationAngle -= 1.0f;

}

if (m\_InputMgr->isKeyDown(VK\_UP))

{

translationZ += 1.0f;

m\_SoundMgr->getSnd("Engine")->playAudio(AL\_TRUE);

}

if (m\_InputMgr->isKeyDown(VK\_DOWN))

{

translationZ -= 1.0f;

m\_SoundMgr->getSnd("Engine")->playAudio(AL\_TRUE);

}

if (m\_InputMgr->isKeyDown(67)) // C Key

{

cameraSwitch = true;

}

if (m\_InputMgr->isKeyDown(86)) // V Key

{

cameraSwitch = false;

}

if (m\_InputMgr->isKeyDown(69)) // E key

{

soundToggle = true;

}

if (m\_InputMgr->isKeyDown(82)) // R key

{

soundToggle = false;

}

It was decided to use a variety of keys to allow control of the player model. The up and down arrow keys where assigned to move the ship forwards and backwards which would also produce an engine sound each time it is pressed. Left and Right rotates the craft around the respective direction.

The C and V key were assigned to deal with the camera switching was earlier discussed in the main class explanation, it is here the Boolean value is altered and then passed.

In a similar fashion the sound keys were assigned to E and R which again affects the sound toggle Boolean value when turns the music theme on or off.

The space bar has been assigned to the control that player uses to fire lasers which gets its firing direction from the models rotation, each laser that is generated is added to a array where its attributes such as speed, scale and direction is assigned, in addition the sound affect is played.

The iterator which was called in the main is assigned here which handles the laser projectiles.

Finally the calculations and variables for the players movements are performed, this includes the rotation angle being changed as well as the velocity generated when moved.

**References**

Books

“Game Coding Complete Fourth Edition” by Mike McShaffry and David “Rez” Graham

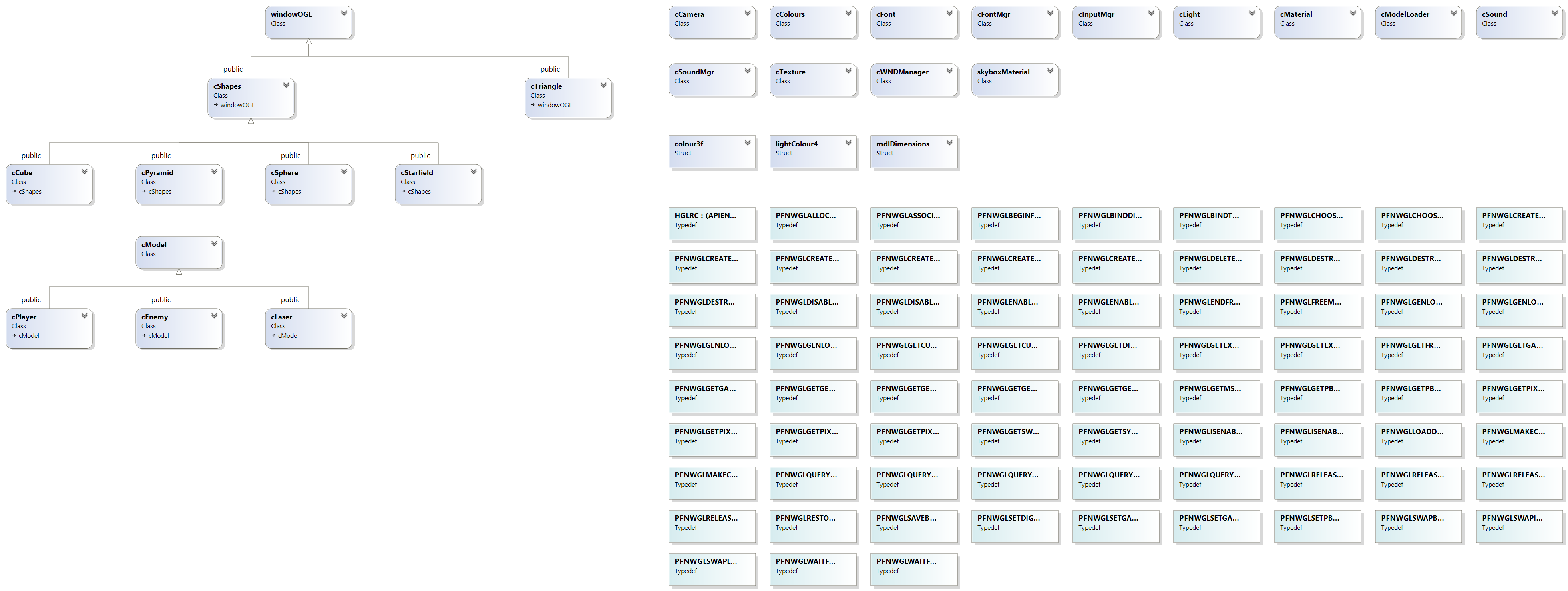
Sites

<https://www.opengl.org/sdk/docs/man2/xhtml/glFog.xml>

OpenGL site for explanation of Fog code

<https://css-tricks.com/snippets/javascript/javascript-keycodes/>

For a reminder of value for input keys

**Class Diagram**

Produced in Visual Studio

**Storyboard**

