Programming Game Engines
ITP485 (4 Units)

Labs

Spring 2010

Version 6.1

Jason Gregory
Lead Programmer, Naughty Dog Inc.

Week 1. Introduction

During the first half of the semester, you will work individually on a PONG-like game, using the OGRE 3D rendering engine. In the second half of the semester, the class will work as a team (or two competing teams) on a larger-scale game project, built on OGRE and a skeletal game engine provided by the instructor.

This section describes how to connect to the Subversion source code repository we'll be using during the semester, how to download and build OGRE, and generally how to get set up. In the OHE lab, most of this set-up work has already been done for you. However, these instructions will be useful if you are setting up your home machine or laptop.

1.1. Setting Up the Source Tree

In the OHE lab, the ITP485 source code tree is located at C:\Document and Settings\All Users\Documents\ITP485\. On your home machine or laptop, you can put the tree anywhere you'd like. For the purposes of this document, the full path will be abbreviated to simply **ITP485**. When first downloaded, the tree should look like this:

`ITP485\`**`Binaries`**`\`	This folder contains a "shippable" binary image of the game(s).
`ITP485\Binaries\Debug\`	Contains the debug executable and DLLs.
`ITP485\Binaries\Release\`	Contains the release executable and DLLs.
`ITP485\Binaries\Media\`	Contains all game assets (meshes, textures, audio, animations, …)
`ITP485\`**`Build`**`\`	The VisualStudio projects are set up to emit intermediate files (.obj, .lib, etc.) here.
`ITP485\`**`Dependencies`**`\`	This folder should contain any 3rd party SDKs upon which the game depends, including OGRE and any other SDKs you decide to use.
`ITP485\Dependencies\ogre\`	This is where OGRE SDK should reside. You must download and build it manually (see below), as it is not included in the distribution.
`ITP485\`**`Engine`**`\`	This folder contains shared engine source code.
`ITP485\`**`GameX`**`\`	This folder contains game-specific code for one student game.
`ITP485\`**`GameY`**`\`	This folder contains game-specific code for a second student game. (At USC we sometimes divided the students into two teams for a bit of friendly competition.)
`ITP485\`**`Students`**`\`	Individual student labs (PONGRE etc.) go here; one subfolder per student.

1.2. Downloading and Installing OGRE

Once you have the *ITP485* lab source tree on your hard drive, you'll need to install and set up OGRE.

- Go to http://www.ogre3d.org and hit the big yellow **DOWNLOAD!** button.
- Select **2. Download a source package** from the OGRE download page.
- Select **OGRE 1.6.3 Source for Windows**, which will direct you to a SourceForge site. Hit **Save File** and save it under ITP485\Dependencies\.

- Select **Visual C++.Net 2008 (9.0) Precompiled Dependencies**, which will direct you to SourceForge again. Hit **Save File** and save it under ITP485\Dependencies\ogre\. (You'll have to create the ogre\ subfolder manually.)
- Use Windows Explorer to navigate to ITP485\Dependencies\.
- Unzip the ogre-v1-6-3.zip file that you downloaded—it should extract into ITP485\Dependencies\ogre\.
- Navigate into ITP485\Dependencies\ogre\.
- Unzip the OgreDependencies_VC9_Eihort_20080203.zip file that you downloaded—it should extract into two subfolders: ITP485\Dependencies\ogre\Dependencies\ and ITP485\Dependencies\ogre\Samples\.

BE CAREFUL TO EXTRACT THESE FILES TO THE CORRECT FOLDERS. The OGRE installation will not interact correctly with the *ITP485* source code if it is not located in *RootFolder*\Dependencies\ogre\ or if its own dependency libs and headers are not located in *RootFolder*\Dependencies\ogre\Dependencies\. In this document, I show *RootFolder* to be ITP485\ as an example, but it can actually reside anywhere.

1.3. Downloading and Installing the DirectX SDK

- Download and install the March 2009 DirectX SDK from http://msdn.microsoft.com/en-us/directx/aa937788.aspx.
- Run MS Visual Studio 2008.
- Select **Options…** from the main **Tools** menu.
- In the scrolling tree view on the left-hand side of the Options dialog, expand the **Projects and Solutions** subtree, and select **VC++ Directories**.
- In the **Show directories for:** drop-down combo box, select **Include files**. Ensure that the DirectX include directory is listed. Use the "up arrow" button to move it to the top of the list.
- In the **Show directories for:** drop-down combo box, select **Library files**. Ensure that the DirectX lib directory is listed. Use the "up arrow" button to move it to the top of the list.

1.4. Building OGRE

To build OGRE, you can simply run **BuildOgre.bat**, located in ITP485\Dependencies\. For this to work, you need to add the path to the Visual Studio IDE's executable to your `%PATH%` environment variable, which you can do by typing:

```
set PATH=%PATH%;C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE
```

at the Windows command prompt. Then **cd** into the ITP485\Dependencies directory and type:

```
BuildOgre
```

to run the script.

You can also build OGRE manually by following these instructions:

- Navigate to ITP485\Dependencies\ogre\. Double-click **Ogre_vc9.sln** (presuming you are using MS Visual Studio 2008).
- Within Visual Studio, go to the **Solution Explorer** window and right-click on any of the sample apps. (I often use the **Skeletal Animation** sample, but any of them will do.) Select **Set as Startup Project** from the pop-up menu.
- At the top of the main Visual Studio window, select **Debug** from the **Solution Configurations** drop-down combo box.
- Right-click your sample app again, and this time select **Build** from the pop-up menu. This will build the OGRE libraries in debug mode, and then build your selected sample app.
- Hit F5 to run the sample app you just built. Verify that it runs properly.
- Right-click the **OgreGUIRenderer** project and build it as well. (It builds a DLL so there's no need to try to run it.)
- Change the **Solution Configuration** to **Release**. Build both your selected sample app and **OgreGUIRenderer** again in Release mode.
- In Windows Explorer, navigate to ITP485\Dependencies\ogre\Samples\Common\bin\Debug\ and copy all the .DLL files you find there into ITP485\Binaries\Debug\. Also copy all the .DLL files from ITP485\Dependencies\ogre\Samples\Common\bin\Release\ to ITP485\Binaries\Release\.

At this point you should be ready to embark on the lab exercises.

Week 2. PONGRE: One-Dimensional Bouncing Ball

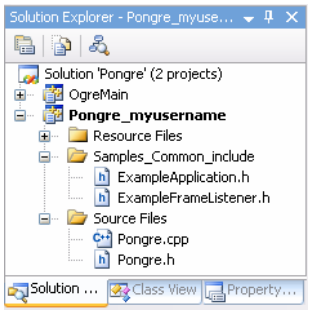
This is the first of three labs in which each student works individually on a simple PONG-like game (which I like to call PONGRE).

2.1. Set Up Your PONGRE Project Folder

- Copy the ITP485\Students\PongreTemplate\ folder and name it ITP485\Students*StudentName*\.
- In Windows Explorer, turn on display of hidden files and folders. (Navigate to the **Tools** menu, **Folder Options…**, and activate the **View** tab. Then make sure the **Show hidden files and folders** radio button is checked. Also *uncheck* the **Remember each folder's view settings** check box, and then hit the **Apply to All Folders** button and the **OK** button.)
- Navigate to ITP485\Students*StudentName*\ and delete the hidden .svn\ folder therein. (This ensures that Subversion will not mistakenly confuse the new folder with the ITP485\Students\PongreTemplate\ folder from which it was copied. You can now safely add ITP485\Students*StudentName*\ to Subversion.)

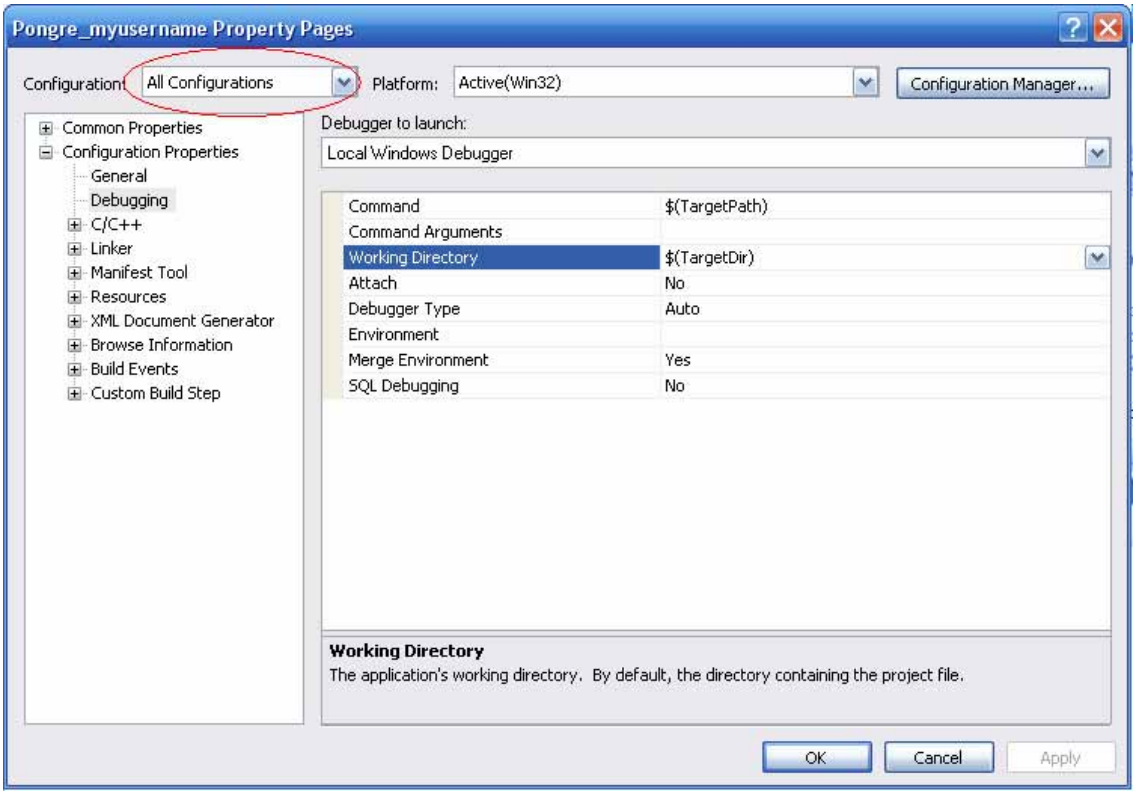
2.2. Configure Project Name and Debugger Settings

- Open your ITP485\Students*StudentName***Pongre.sln** in Visual Studio 2008.
- Expand the top-level **Pongre** solution, located in Visual Studio's **Solution Explorer** tree view. Click on the **Pongre_myusername** project beneath it, and hit F2 and *rename* it to something unique like "Pongre_abc" (where "abc" is your initials or some other unique identifier). This will help to ensure that the students' Pongre.exe files do not conflict with one another in the ITP485\Binaries\Debug\ and Release\ folders.



- Right-click the (newly renamed) Pongre_abc project, and select **Properties**.
- IMPORTANT: Select **All Configurations** from the **Configuration:** combo box along the top of the window. (If you don't do this, you'll be changing the Debug configuration but you'll be leaving the Release configuration incorrect.)
- Select the **Configuration Properties**|**Debugging** tab from the tree view on the left.
- Make sure the **Command** field contains `$(TargetPath)`.

- Change the **Working Directory** field to `$(TargetDir)`.
  THIS IS CRUCIAL. It allows OGRE to find its DLLs and engine configuration files when the game runs. (If you get errors about the game failing to load DLLs or .cfg files, double-check that your working directory is set correctly.)
- Click the **Apply** button.



- In Pongre.h, find the following line of code:

```
// Add my private Media folder to OGRE's default resource path.
ResourceGroupManager::getSingleton().addResourceLocation(
   Ogre::String("../../Students/myusername/Media"),
               // IMPORTANT: Replace this with your user name!
   Ogre::String("FileSystem"),
   Ogre::String("General"),
   true); // recursive - add all child folders as well
```

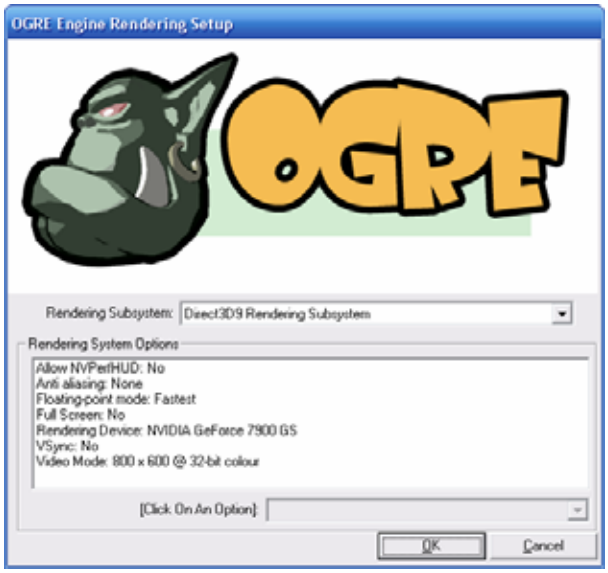
  Replace "myusername" with the name of your student folder. For example, if your student folder is ITP485\Students\JoeBlow\ then the edited code should look like this:

```
// Add my private Media folder to OGRE's default resource path.
ResourceGroupManager::getSingleton().addResourceLocation(
   Ogre::String("../../Students/JoeBlow/Media"),
   Ogre::String("FileSystem"),
   Ogre::String("General"),
   true); // recursive - add all child folders as well
```

- Now save everything by selecting **Save All** from the **File** menu on the menu bar.

2.3. Run PONGRE for the First Time

- Build and run your PONGRE game at least once to familiarize yourself with everything.
- When PONGRE runs, you should see the OGRE configuration dialog box.



- Select the **Direct3D9 Rendering Subsystem** from the combo box.
- Click on **Full Screen:** and select **No** from the **[Click On An Option]:** combo box. IMPORTANT: Always run in *windowed mode* during development. Debugging is a lot easier when you can flip back and forth freely between your game and Visual Studio.
- Click OK, and after a few seconds your nascent PONGRE game should run!

2.4. Set Up the Camera

The code generated by the OGRE app wizard allows the camera to be flown around in 3D via the keyboard and mouse. For PONGRE, we'll want to disable this feature so that our camera always looks straight down the world *z* axis (to simulate the 2D look of the original PONG). We'll also want to pull the camera back a bit.

2.4.1. Position the Camera

- In Pongre.h and PongreFrameListener.h, you should find two classes: `PongreFrameListener` and `PongreApplication`.
- In the `PongreApplication` class, you'll find a virtual override of the function `createCamera()`.

```
void PongreApplication::createCamera(void)
{
      // Create the camera
      mCamera = mSceneMgr->createCamera("PlayerCam");

      // Position it at 500 in Z direction
      mCamera->setPosition(Vector3(0,0,500));
      // Look back along -Z
      mCamera->lookAt(Vector3(0,0,-300));
      mCamera->setNearClipDistance(5);
}
```

- Change the camera's *z* coordinate from 500 to **300** (or position it to suit your own taste).

2.4.2. Disable Camera Controls

- By default, the `ExampleFrameListener` class provides us with mouse- and keyboard-driven camera controls.
  - The ExampleFrameListener.h header file can be found at **ITP485\Dependencies\ogre\Samples\Common\include**.
- The relevant bits of `ExampleFrameListener` have been duplicated into our `PongreFrameListener` class so that we can modify them freely. Open the PongreFrameListener.cpp file, so you can see how it works its magic.
  - The main game loop is implemented in a virtual callback function named:
    - `frameRenderingQueued()`

    It is called automatically by the OGRE example application framework, right after the scene's rendering commands have been queued up for execution by the GPU. (This is a good time to run game logic for next frame, because it will run in parallel with the GPU.)
  - In this function, you'll find calls to two functions that poll the keyboard and mouse, and translate these inputs into potential camera movements:
    - `processUnbufferedKeyInput()`
    - `processUnbufferedMouseInput()`
  - Ultimately, however, the camera movements are not *applied* until the following function is called, at the end of `frameRenderingQueued()`:
    - `moveCamera()`
- So, to disable camera movement, we simply need to modify our override of the `moveCamera()` function, and arrange for it to do nothing:

```
void PongreFrameListener::moveCamera()
{
   // camera movement disabled for PONGRE
}
```

- Build and run Pongre.exe. You should find that you can no longer fly the camera around.
  - **BONUS:** Allow camera movement controls to be turned on and off via a key press. (The ability to fly the camera around can be useful for debugging.) Be sure to set the camera back to the correct position and orientation for PONGRE gameplay when the movement controls are turned back off.

2.5. Make the Ball Bounce in One Dimension

Our "ball" is going to be represented by the OGRE head mesh. In this section of the lab, we're going to animate the ball bouncing back and forth in one dimension between two imaginary planes.

In OGRE, a triangle mesh is represented by an object of type `Ogre::Mesh`. Each mesh exists once in memory, even if there are many instances of that mesh on-screen. The class `Ogre::Entity` represents one *instance* of an `Ogre::Mesh`.

In order for OGRE to render a mesh instance (i.e., an `Ogre::Entity`), it must be wrapped in a scene node (`Ogre::SceneNode`). The scene node's job is to maintain the mesh instance's **position** and **orientation**, and to link it into OGRE's scene graph. So, to move the OGRE head mesh (our "ball") around in world space, we must interact with its `Ogre::SceneNode`.

Look in `PongreFrameListener::createScene()` and you'll see the `Ogre::Entity` and `Ogre::SceneNode` being created.

2.5.1. Write the moveBall() Function

- In order to move the ball around, add a new function to your `PongreFrameListener` declared as follows:
  - `void moveBall(const FrameEvent& evt);`
- Add a call to `moveBall()` at the end of `PongreFrameListener::frameRenderingQueued()`.
- To interact with the `Ogre::SceneNode` created in `createScene()`, we'll need a pointer to it. Change the local variable `ogreHeadNode` into a *member variable*. (I recommend renaming it `mOgreHeadNode` or `m_ogreHeadNode`. The 'm' prefix provides a clear indication that the variable is a member of the class.)
- In your `moveBall()` function, use `mHeadNode.getPosition()` and `mHeadNode.setPosition()` (or `mHeadNode.translate()`) to move the ball along the world-space *y* axis.

- The ball should move at a **constant speed** in *game world units per second*. (A speed of 20 units per second works well.)
- Translate this speed into the distance to move per *frame* by multiplying it by the amount of time that has passed since the last call to `frameStarted()`. This quantity is available in the `timeSinceLastFrame` data member of the `Ogre::Event` that you passed to `moveBall()` from `frameRenderingQueued()`:

```
Real speedUnitsPerFrame = speedUnitsPerSecond * evt.timeSinceLastFrame;
```

2.5.2. Make the Ball Bounce

- Check for the ball's *y* coordinate getting larger than predefined minimum and maximum values (`+100.0f` and `-100.0f` work well), and reverse the direction of the ball (negate the sign of the speed) whenever it passes these boundaries.
  - **BONUS:** Apply the distance by which the ball "overshot" the boundary to the ball's motion in the opposite direction, to avoid slight "hitches" whenever the ball bounces off the wall.

2.6. Draw Boundary Walls

PONG traditionally has two stationary boundary walls at the top and bottom of the playfield. The code you wrote above allows the ball to bounce (in one dimension) between the top and bottom limits. Now we'll add the visual representation of the walls by creating two "prefab" plane meshes.

- In `createScene()`, add the following lines:

```
// Create two planes for the upper and lower boundary walls.

Entity* upperWall = mSceneMgr->createEntity("UpperWall",
                                    SceneManager::PT_PLANE);
SceneNode* upperWallNode = mSceneMgr->getRootSceneNode()
                                    ->createChildSceneNode();
upperWallNode->attachObject(upperWall);
upperWallNode->rotate(Vector3::UNIT_X, Radian(Degree(90.0f)));
upperWallNode->setPosition(Vector3(0.0f, 100.0f, 0.0f));

Entity* lowerWall = mSceneMgr->createEntity("LowerWall",
                                    SceneManager::PT_PLANE);
SceneNode* lowerWallNode = mSceneMgr->getRootSceneNode()
                                    ->createChildSceneNode();
lowerWallNode->attachObject(lowerWall);
lowerWallNode->rotate(Vector3::UNIT_X, Radian(Degree(-90.0f)));
lowerWallNode->setPosition(Vector3(0.0f, -100.0f, 0.0f));
```

- This code creates two "prefab" plane meshes, and attaches them to the scene. It also rotates them so they are horizontal (by default plane meshes are created in the *xy* plane),

and slides one up and one down so they are at +100.0f and –100.0f units along *y* respectively.

- **BONUS:** You'll notice that the OGRE head now interpenetrates the walls as it moves up and down. Why is this happening? What simple thing could you do to make the OGRE head appear to collide properly without interpenetration? Fix it.
- **BONUS:** Search through the OGRE samples, and figure out how to add a *textured material* to your walls.

Here's what the final result should look like:

Week 3. PONGRE: Bouncing and Paddles

In this lab, we'll get our OGRE head/ball bouncing properly using vector math. We'll also add some movable paddles, and get the basic PONG-like gameplay functioning.

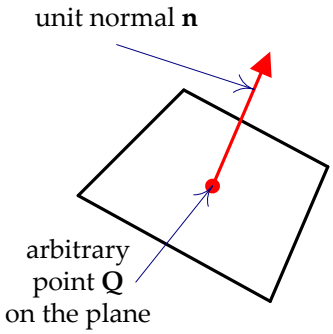
3.1. Two-Dimensional Velocity

- Add a member variable of type `Ogre::Vector` to your `PongreFrameListener` class. This vector will describe the velocity of the ball.
  - Constrain the velocity to the world-space *xy* plane by setting the *z* coordinate of the vector to zero.
- Convert the velocity vector (which should be measured in game units per *second*) into a velocity in game units per *frame* by multiplying the vector by the scalar `evt.timeSinceLastFrame`, as we did in one dimension in Lab 1.
- Apply the velocity vector to the position of the ball each frame.
- Provide a randomized initial velocity to the ball when the scene is first created.
  - The `PongreFrameListener` contains a member object named `mRandMT`. It is a "Mersenne Twister" random number generator, borrowed from the ITP485 sample game engine, and implemented by Shawn Cokus. Use `mRandMT.randomFloatMT()` to generate a random `float` between `0.0f` and `1.0f`.
  - The initial velocity should always have the same fixed *magnitude* (e.g. 50 units/sec), but its *direction* should be randomized. Remember to keep it constrained to the *xy* plane.
- When this is done, your ball will fly off in a random direction… but if your *y*-coordinate collision code is still in place from Lab 2, it should still bounce between the upper and lower infinite planes.

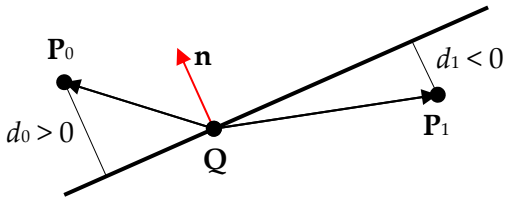
3.2. Bouncing with Vector Math

- Replace your simple *y*-axis collision detection code from Lab 1 with a collision detection routine that can handle **arbitrary planes**.
  - We'll still only use two planes, upper and lower.
  - But now your code should allow the planes to be *arbitrarily* positioned and oriented.
  - To prove that your math works, we'll orient the planes at some non-right angle in world space (e.g. 30 degrees off horizontal) and demonstrate that the collision detection and bounce still works properly.

- Define each plane using an **arbitrary point on the plane** and a **unit normal vector**. (Use `Ogre::Vectors` for both—or you can use an `Ogre::Plane` instead if you prefer.)

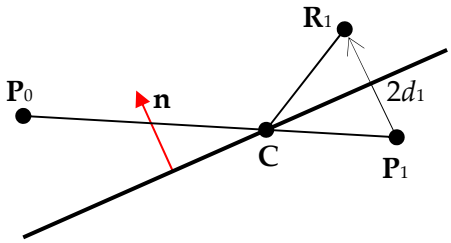


- Determine if the ball has crossed the plane (collided with it) by checking the *distance from the ball to the plane* using a **vector subtraction** followed by a **dot product**.
  - Subtract the position of the ball from the arbitrary point on the plane to get a vector from the point on the plane to the ball.
  - Then take the dot product of this vector with the plane's normal vector. The resulting dot product is the height of the ball above or below the plane.
- Calculate the dot product for the position of the ball last frame ($\mathbf{P}_0$), and for the position this frame ($\mathbf{P}_1$).
  - $d_0 = (\mathbf{P}_0 - \mathbf{Q}) \bullet \mathbf{n}$
  - $d_1 = (\mathbf{P}_1 - \mathbf{Q}) \bullet \mathbf{n}$
  - If the dot product last frame d_0 is positive but the dot product this frame d_1 is negative, the ball has crossed (collided with) the plane.



- Now, the reflection of point $\mathbf{P}_1$, which we will call $\mathbf{R}_1$, is found by moving $\mathbf{P}_1$ from one side of the plane to the other along the plane's normal vector **n**.
  - The distance to move is simply $2d_1$, since moving it by d_1 would bring it onto the plane, and moving it by another d_1 puts it the same distance on the *front* side of the plane as it was *behind* the plane.

$$\mathbf{R}_1 = \mathbf{P}_1 + (2d_1)\,\mathbf{n}$$

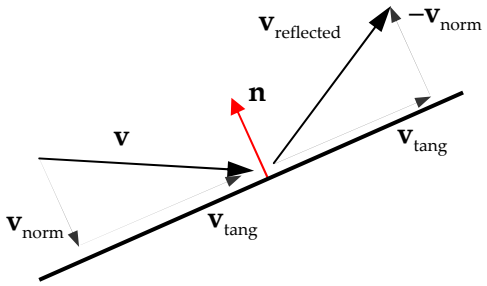


- You'll also have to reflect the ball's velocity vector about the plane. This is done by breaking the velocity into its normal and tangential components, negating the normal component, and then reconstructing the reflected velocity vector from its components:

$$\mathbf{v} = \mathbf{v}_{norm} + \mathbf{v}_{tang}$$

$$\mathbf{v}_{norm} = \mathbf{v} \cdot \mathbf{n}$$
$$\therefore \mathbf{v}_{tang} = \mathbf{v} - \mathbf{v}_{norm} = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})$$

$$\mathbf{v}_{reflected} = \mathbf{v}_{tang} - \mathbf{v}_{norm}$$
$$= [\mathbf{v} - (\mathbf{v} \cdot \mathbf{n})] - (\mathbf{v} \cdot \mathbf{n})$$
$$\therefore \mathbf{v}_{reflected} = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})$$



3.2.1. Account for Ball Radius

- You've probably noticed that the ball embeds itself into the wall before it bounces. This is because we're colliding and reflecting the center point of the ball with the planes.
- To account for the radius of the ball, simply **move the planes inward** along their **normal vectors** by a distance equal to the ball's **radius**.
  - Do not move the *visual* representations of the two planes, only the *mathematical* representation used by your collision/bounce routine.
  - This provides the visual illusion of the edge of the ball hitting the plane, when mathematically we're really still just colliding the center of the ball.
- You can determine the radius of the ball by calling the `Ogre::Entity::getBoundingRadius()` function on the ball entity.

3.2.2. Show That It Works

- Change your two planes so that they are oriented arbitrarily, but still parallel. (Or if you like, make them non-parallel and see what happens.)
- Run your new collision/bounce code and demonstrate that the ball still bounces correctly off the planes, even when they are not axis-aligned.
    - **BONUS:** Provide a way for the user to move and rotate the planes in real-time by pressing keys on the keyboard or via the mouse.

3.3. Add Paddles

- Add two more plane meshes, this time for the movable paddles. Position and orient them appropriately so they appear at the left and right edges of the playfield.
- Shorten the two paddle walls to be about ½ to ¼ the height of the playfield.
- Add code to read the joypad, mouse and/or keyboard in order to adjust the *y* position of each paddle.
- Constrain the *y* motion of each paddle so that its top edge never passes beyond the top wall, and its bottom edge never passes below the bottom wall.
    - **BONUS:** Support *two* human players controlling opposite paddles.

3.3.1. Detect Paddle Collisions

- Change your collision detection code so that when the OGRE head collides with the left or right boundary plane, the *y* position of the point of impact is checked against the position and height of the paddle.
    - If the impact point lies within the vertical extents of the paddle, allow the OGRE head to bounce, as it does when it intersects the top and bottom wall planes.
    - If it does not lie within the paddle's extents, allow the head to pass through the plane.
    - HINT: This is probably a good time to refactor your code to introduce a `Wall` class and a derived `Paddle` class.
- It's OK to assume that your paddles are always perfectly vertical (*i.e.* to use a simple check of the *y* coordinate to determine if the ball is within the boundary of the paddle).
    - **Extra credit:** If you want to practice your 3D math, try determining the actual point of impact in 3D and then doing a test for whether the point is within the rectangular bounds of your paddle. Then allow the entire playing field (all four planes) to be oriented at an arbitrary, non-axis-aligned angle and show that it still works.

3.3.2. Handle Misses

- We'll add scoring and a heads-up display later. For now, if the ball misses one of the paddles, just allow it to continue for a short distance in the *x* direction past the wall, then reset the ball to the center of the playing field with a new randomized initial velocity.

Week 4. PONGRE: Congratulations… It's a Game!

In this lab we'll finish the core gameplay features by adding scoring, a heads-up display, and improved graphics.

4.1. Add Required Features

The following features are required in some form:

4.1.1. Scoring

- Keep track of the two players' scores, and display them on-screen using Ogre's Overlay system.
- The overlays used to display these stats are defined in an overlay script called OgreDebugPanel.overlay, which can be found in the ITP485\Binaries\Media\packs\OgreCore.zip file.
- Extract OgreDebugPanel.overlay to your **Students*myusername*\Media** folder, rename it to PongrePanel.overlay, and then edit it to contain the panel(s) you want.
- To learn the basics of the Overlay system, take a look first at how ExampleFrameListener.h from the Ogre Example Framework displays the Average FPS and other statistics on-screen.
- You can also have a read through the following Ogre Wiki snippet (http://www.ogre3d.org/wiki/index.php/Creating_Overlays_via_Code) to learn more about creating overlays programmatically instead of using a .overlay script.

4.1.2. Improve the Graphics

- Change your paddles from simple flat planes into some kind of 3D mesh.
  - You can find OGRE format **.mesh** files in the ITP485\Binaries\Media\models\ folder. You may also be able to find some OGRE meshes online.
- You may leave the upper and lower walls as simple planes, but try modifying the material used by your planes so they have a texture. Texture (image) files can be found in ogre/Samples/Media/materials/textures.
- **IMPORTANT:** You should store all of your assets in your **Students*myusername*\Media** folder. Do *not* check new assets into the global Binaries\Media\ folder, as these assets might interfere with other students' PONGRE projects or the class-wide game project.

Week 5. PONGRE: Final Polish

Use this final week to finish up any aspects of your game that are incomplete. If you have time, customize your PONGRE to set it apart from the other students' games.

5.1. Add Creativity and Polish

Here are some ideas for adding creativity and polish to your PONGRE. You are, of course, encouraged to generate your own ideas as well. Choose **one or two** features at most to implement, and do so *only* once you've completed the basic requirements of the project. Do not bite off more than you can chew! One custom feature implemented well is far superior to three features implemented poorly.

- When one player reaches a pre-defined score, declare that player the winner, then re-start the game.
- Use Microsoft XACT to play simple sound effects at appropriate moments in the game.
- Render a textured plane that covers the entire screen, and use it as a "splash screen" for your game that shows the title and your name. Keep it up for a few seconds, then activate the core gameplay.
- Tennis cam – have the camera yaw slightly to follow the ball as it bounces back and forth. (But be careful not to create a "nausea cam!")
- Do some kind of cool camera animation whenever one player scores a point on the other. Maybe zoom in on the opponent's paddle, or rotate 360 degrees around the play field, or whatever you can dream up!
- Add a single-player mode by implementing some simple A.I. to control the second paddle.
- Permit walls to be angled. For example, one player could be given a handicap by widening his or her end of the play field so there's more goal space to cover.
- How can you make your PONGRE game more three-dimensional?

Week 6. Class Project: Game Design

For the remainder of the semester, your entire class will work together as a team to build a game with a much larger scope than your individual PONGRE projects. (If enough students are available, the class may be split into two or more game teams for a little friendly competition.)

The class project involves building a simple "platformer" game similar to the bonus levels in *Super Mario Sunshine*.

The game must be implemented with 3D graphics using OGRE. The gameplay can either be fully three-dimensional, or it can be primarily two-dimensional, as it is in games like *Little Big Planet* or *New Super Mario Bros. Wii*.

- Bear in mind that your game design should be *much simpler* than even the simplest bonus levels in *Super Mario Sunshine.* (You won't have time to do anything more ambitious.)
- The key to success is to *start small*, adding more-advanced features as and *only if* you have time.

The deliverable for this week's lab is a **concise game design document** for your game.

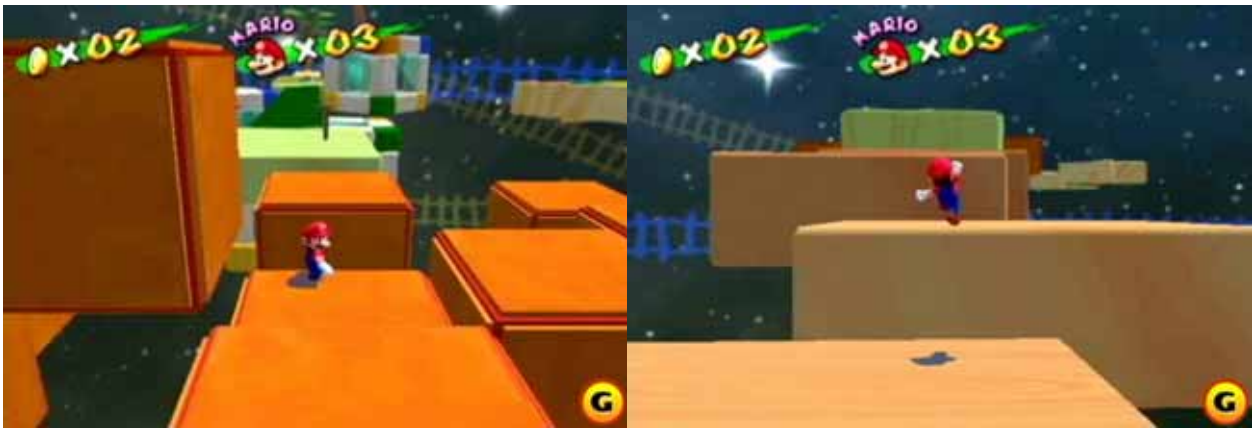
6.1. Game Design

You are free to design your game as you see fit, as long as the design is approved by the instructor on the basis of feasibility and pedagogical value. That said, what follows is a sample game design document that is suitable for a 13-week schedule.

6.1.1. Game World

- The game world consists primarily of movable, strictly horizontal, bounded rectangular planes (the top surfaces of the blocks in the above diagram).
  - Platform blocks hover in mid-air.
  - An infinite horizontal "death plane" lies beneath all platforms, so that if the player falls off a block he will fall to his death.
  - Platform blocks will start out stationary. Movable blocks, blocks that disappear and reappear, *etc.* may be added later, time permitting.

6.1.2. Player Mechanics

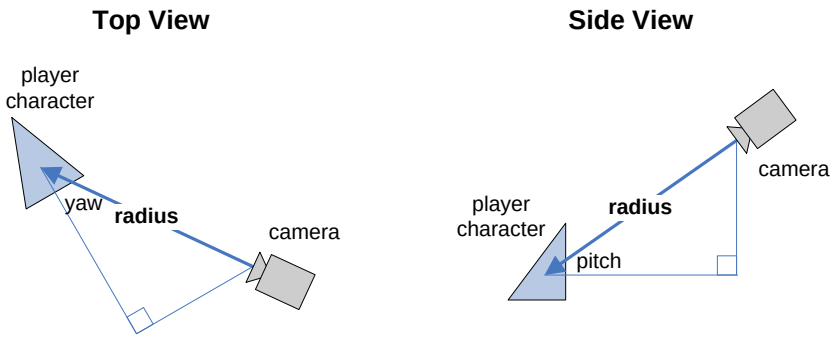


- The core player mechanic consists of sliding about on a plane under joypad control.

- The player character's body will appear to move using skeletal animations provided by the instructor. (We are using the *Metalhead Flut-Flut* character from *Jak 3*, used with permission from Naughty Dog, Inc.)
- However, in reality he'll just be sliding around on the plane.

- The player character can also jump in the air.
  - Jump follows a simple parabolic path.
  - If the path collides with another platform, the player "acquires" the new platform and goes back into "slide along plane" mode.
- The player character can slide off the edge of a platform and start falling.
  - Falling or jumping may land the player character on another platform.
  - Otherwise, the player is killed when it passes through an infinite "death plane" located below all other platforms.
  - Upon death, the player is re-spawned at the start of the level, at the most recent check-point, or on last the platform he was on when he fell.

6.1.3. Camera



- The game camera is fundamentally a simple look-at camera, with the player at the target point.
  - The camera tries to maintain an ideal distance behind and height above player character.
  - The camera's world-space position/orientation is on "springs" so as the player moves, the camera "catches up" rather than rigidly following the player.
    - Instructor will provide code for a critically-damped spring.
  - Normally the camera tries to stay directly behind the player; however, the user has control of camera yaw via the right joypad stick.

6.1.4. Enemies/Obstacles

- The player should face some kind of obstacle(s) and/or enemies in the game world.
- This could be a race against time, hazards on the playfield, or AI controlled NPCs.

6.1.5. Audio

- Simple audio will be developed for the game, with help of Microsoft XACT.
- Audio features may include:
  - sound effects for player walking, jumping, landing, changing direction rapidly (180 degree turns), etc.
  - background music,
  - special music for when player falls to his death,
  - possibly sound effects for moving platforms, obstacles, etc.

6.2. Teams

The entire class will work together on a single game project. Or, if enough students are available, the class may be split into two or more game teams for a little friendly competition.

The structure of each team will closely match that of a real engineering team in the game industry. As such, each team will consist of small (1–2 person) sub-teams focused on (at least) the following technology areas:

- **Player** (1–2 engineers, at least one senior)
  - player movement on platforms (working closely with Environment team)
  - jumping and falling
  - character animation
  - action state machine (locomotion, jumping, falling, dying, attacking, etc.)
- **Environment** (1–2 engineers, at least one senior)
  - platforms and any other static "background" scene elements
    - rendering
    - collision detection
- **Camera** (1–2 engineers, one senior recommended)
  - design and implement player "follow cam"
  - additional camera modes/types if and as time permits
  - support debug "fly through" camera as needed
- **Hazards/Enemies** (1–2 engineers)
  - design and implement playfield hazards, and/or AI-controlled enemy non-player characters
  - optionally work with Environment team to design and/or implement specialized platform movement modes (moving, shrinking, disappearing) and/or surface types (e.g., ice=slide, mud=slow down, fire=damage, etc.)
  - work with Player team to ensure proper interaction between hazards/enemies

- **Effects** (1–2 engineers)
  - add suitable visual effects to static and dynamic scene elements as appropriate
  - light the game world(s)
  - set up the sky box (if called for by the game design)
  - implement splash screen, HUD and in-game menus
  - add sound effects and music using the Microsoft XACT audio tool and SDK

6.3. Game Assets Provided

Naughty Dog Inc. has graciously donated the **Metalhead Flut-Flut** character model from *Jak 3* for use in student projects. The model is comprised of a textured triangle mesh, a skeleton, and a full set of animations. You are encouraged to use this model as the main player character in your student games. You are free to provide your own animated model(s) if you prefer, but be aware that the model will need to be of a sufficiently high quality, and must have a suitable set of animations.

The following animations are provided for the Metalhead Flut-Flut character:

- ambient
- ambient-scratch
- bounce-back
- death
- diving-attack
- double-jump
- flut-death
- head-attack
- hit-back
- idle
- idle2
- jog
- jump
- jump-forward
- jump-loop
- kanga-catch
- run
- walk

The names of the joints in the Flut-Flut skeleton are as follows:

- 000: flut__jnt_main
- 001: flut__jnt_hips
- 002: flut__jnt_Lthigh
- 003: flut__jnt_Lquarter
- 004: flut__jnt_Lknee
- 005: flut__jnt_Lankle

- 006: flut__jnt_lInToe
- 007: flut_lInToeEND
- 008: flut__jnt_lOutToe
- 009: flut_lOutToeEND
- 010: flut__jnt_lFrontToe
- 011: flut_lFrontToeEND
- 012: flut_LfootEND
- 013: flut__jnt_lRearToe
- 014: flut_lRearToeEND
- 015: flut__jnt_Rthigh
- 016: flut__jnt_Rquarter
- 017: flut__jnt_Rknee
- 018: flut__jnt_Rankle
- 019: flut__jnt_rInToe
- 020: flut_rInToeEND
- 021: flut__jnt_rOutToe
- 022: flut_rOutToeEND
- 023: flut__jnt_rFrontToe
- 024: flut_rFrontToeEND
- 025: flut_RfootEND
- 026: flut__jnt_rRearToe
- 027: flut_rRearToeEND
- 028: flut__jnt_tail1
- 029: flut__jnt_tail2
- 030: flut__jnt_tailFL
- 031: flut_tailFLEND
- 032: flut__jnt_tailFR
- 033: flut_tailFREND
- 034: flut__jnt_tail3
- 035: flut_tailEND
- 036: flut__jnt_chest
- 037: flut__jnt_neck
- 038: flut__jnt_head
- 039: flut_headEND
- 040: flut__jnt_ploom1
- 041: flut__jnt_ploom2
- 042: flut__jnt_ploom3
- 043: flut_ploomEND
- 044: flut__jnt_jaw
- 045: flut_jawEND
- 046: flut__jnt_tongue
- 047: flut__jnt_reign_l1
- 048: flut__jnt_reign_l2
- 049: flut__jnt_reign_l3
- 050: flut__jnt_reign_l4
- 051: flut__jnt_reign_lend
- 052: flut__jnt_reignr1

- 053: flut__jnt_reignr2
- 054: flut__jnt_reignr3
- 055: flut__jnt_reignr4
- 056: flut__jnt_reignr5
- 057: flut_reign1
- 058: flut_reign2
- 059: flut_reign3
- 060: flut_reign4
- 061: flut_reignEnd
- 062: flut_reignL4
- 063: flut_reignR4
- 064: flut_reignL3
- 065: flut_reignR3
- 066: flut_reignL2
- 067: flut_reignR2
- 068: flut_reignR1
- 069: flut_reignL1
- 070: flut_eyeLeftJoint
- 071: flut_eyeRightJoint
- 072: flut__jnt_lShoulder
- 073: flut__jnt_lWing1
- 074: flut__jnt_lWing2
- 075: flut__jnt_lWingF2
- 076: flut_lWingF2END
- 077: flut_lWingEND
- 078: flut__jnt_rShoulder
- 079: flut__jnt_rWing1
- 080: flut__jnt_rWing2
- 081: flut_rWingEND
- 082: flut__jnt_rWingF2
- 083: flut_rWingF2END
- 084: flut_jak_bak

Weeks 7–13: Approximate Project Schedule

For the remainder of the semester, project sub-teams should do their best to meet the milestones laid out in the following pages. This is an approximate schedule only. Teams should feel free to adjust the sequence of milestones and/or the specifics of each milestone as they see fit to accommodate the game design and the teams' development style. The only hard deadlines are the **Gold Master** lock-down, which will occur **one week prior** to the project due date, and the final due date of the project(s) at the end of the semester.

7.1.1. Gold Master

On the day of Gold Master lock-down, all sub-teams should submit their final work to the Subversion server, and then cease development. During Gold Master week, everyone should play the game as much as possible, looking for critical bugs. If a bug is found, the lead engineer, teaching assistant(s) and the instructor must be notified. Together, they will decide whether the bug is severe enough to warrant the risk of fixing it. If a fix is approved, the team member most familiar with the code involved will fix the bug and submit it to Subversion. On the day prior to the due date, the lead engineer, TA(s) and/or instructor will do a final "sync" of the Subversion repository, and build the final version of the game(s). This build will be used for evaluation, and possibly also for public demonstration of the game(s) on the last day of class.

7.2. Player Team

7.2.1. Preexisting Functionality

To save time, and to ensure that all teams can hit the ground running, a very simple player character mechanic has already been implemented for you. It is functional, but in no way polished. The code is located in the ITP485\GameX\Player\ folder (and duplicated in ITP485\GameY\Player\). The player's rather minimal feature set includes:

- The ability to slide around on the $y = 0$ plane, under joypad control.
- The ability to jump, fall, and land back on the $y = 0$ plane.
- Controls are camera-relative, using the **debug fly-through camera** as its reference frame, since the **follow cam** will not yet be ready. Here's how the controls work:
  - The left joypad stick provides a raw 2D control vector, (x_{joy}, y_{joy}).
  - This vector is converted into a 3D **camera-space** control vector $(x_{cam}, y_{cam}, z_{cam}) = (x_{joy}, 0, y_{joy})$.
  - The camera-space vector is then transformed into a **world-space** control vector $(x_{world}, y_{world}, z_{world})$ by multiplying it by the camera-to-world transform.
  - Finally the world-space control vector is transformed into **player space**. (Player space is a coordinate frame whose z axis points in the direction the player is facing, with x to the left and y up.)
  - The z component of the player-space control vector defines how much the player should **accelerate** or **decelerate** along its current "forward" axis.
  - The x component of the player-space control vector defines how much the player will tend to **rotate** about its "up" axis, in an attempt to align its "forward" axis with the world-space control vector.
  - The x component is scaled to generate a suitable rotational speed ω (measured in radians/sec). This speed is converted into a rotation delta by multiplying it by the frame delta time ($\Delta\theta = \omega \Delta t$). This is called "explicit Euler" numerical integration. The rotation delta is applied to the orientation of the player character each frame.
- The Player has a simple **finite state machine** to manage its states (sliding on a surface, falling, launching into a jump, etc.)
  - Notice that the Player's `Update()` function calls a state-specific update function, depending on which state the Player is currently in.

7.2.2. Week 7

- Smooth and tweak the linear and angular acceleration/deceleration of the character until the player controls "feel good."

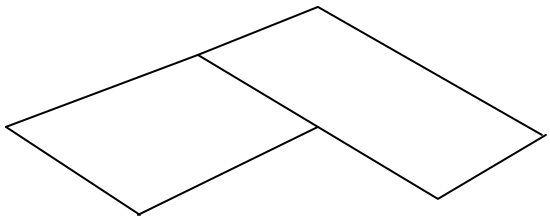
- Add animation to your player character by calling its `playAnimation()` function (inherited from `MeshInstance`) at appropriate locations in the code. See section *6.3 Game Assets Provided* for a list of the animations that are available for the Flut-Flut.
- Each player state typically corresponds to a particular kind of animation:
  - While locomoting on a platform surface, you can blend between walk, jog and run, or just play a run animation at all times.
  - Also add animations for when the player is launching into a jump, flying through the air, and so on.

7.2.3. Week 8

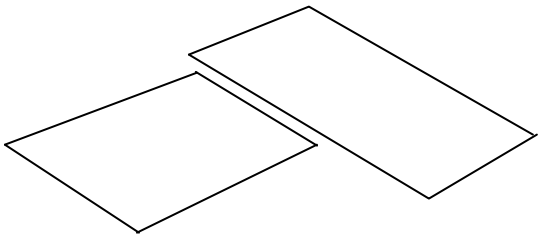
- Work with the Environment team to position the player character on a starting platform. (This can be as simple as selecting a suitable y coordinate, assuming the platforms are always perfectly horizontal.)
  - While on a platform, the player controls can work just as they did on the $y = 0$ plane last week.
- Keep track of the player's current platform. Detect when the player has passed over the edges of the platform and should begin to fall.
  - Implement falling by maintaining the player's horizontal (xz) velocity, and applying gravitational acceleration to the velocity's y component via simple Euler integration. The player should trace out a parabolic path.
- During each frame while the player is in the falling state, the platform system should be queried to determine whether a new platform has been struck. If so, attach the player to the new platform and transition it back into the "sliding around on a platform" state.

7.2.4. Week 9

- Add in-air control to the player.
- Tweak the play-back rate of walk/run animations to reduce foot sliding.
- Add code to handle the following special cases:
  - Walking from one platform to another immediately adjacent platform with an identical y coordinate.



  - Walking from one platform to another immediately adjacent platform that is slightly higher than the current platform (walking up a "curb").



- OPTIONAL: Stopping the player from moving when it encounters an adjacent platform that is too high to walk up automatically, but too low to pass underneath.

7.2.5. Weeks 10–14

- Add player "death plane" beneath all platforms.
  - When player dies from a fall, respawn at the last checkpoint or on the last "good" platform.
- Continue to fine-tune the player mechanics.
  - Work with the Camera team to fine-tune the follow cam and its effects on the player's controls. (The follow camera and player mechanics can and should be prototyped separately, but you'll find that each one's parameters can have profound effects on the other. Hence final tuning of each must be done in tandem with the other.)
- Work with the Environment and Hazards/Enemies teams to implement power-ups, platform surface types, reactions to hazards, attacks and hit reactions, etc.
- Add a double-jump if you feel it enhances gameplay.
- Add support for moving platforms, shrinking platforms, tilting platforms, etc. if *and only if* you have time, and the *core mechanics* are working flawlessly. (And only if the Environment team also has bandwidth to tackle this.)

7.3. Environment Team

7.3.1. Overview

- The Player team starts with a simple mechanic allowing the player character to slide around on an infinite horizontal plane.
- While the Player team is developing its initial functionality, the Environment team should decide on the kind of game world they will provide.
  - If the recommended game design outlined in this document is used, then the game world will consist of a collection of axis-aligned rectangular platforms.
  - Other possibilities include movable platforms, horizontal but not necessarily axis-aligned platforms, spherical platforms as seen in Super Mario Galaxy, arbitrary non-horizontal platforms, perhaps implemented in pseudo-2D as seen in Little Big Planet, etc.
  - For the remainder of this section, we will assume the game design calls for axis-aligned horizontal rectangular platforms.
- The **collision geometry** of each platform can be represented by a simple 3D axis-aligned bounding box (AABB), i.e. (x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}). For the most part, the player character will only ever interact with the top xz plane of this AABB, but it's a good idea to use a full 3D AABB to make more-advanced collision detection easier later on.
- The **renderable geometry** of each platform should be an OGRE-compatible 3D triangle mesh. For starters, you can use OGRE's "prefab" box mesh. However, to make your game world more visually interesting, you'll probably want to find or create some better-looking meshes. Remember, even though the collision representation of each platform is a simple AABB, your visible geometry needn't conform exactly to this shape. The platforms might look like organic slabs of rock and dirt that are only roughly rectangular. Or they might resemble carved gem stones, or slabs of ice, or futuristic metallic hover-pods… let your imagination run wild.

7.3.2. Week 7

- Decide on the kind of game world you will provide – how will the collision geometry be represented?
- Implement a `Platform` class, derived from `MeshInstance` (so it can be easily rendered as a triangle mesh). Set it up to contain your chosen collision geometry representation (e.g. 3D AABB). It's probably a good idea to create a subfolder of your GameX\ or GameY\ folder called Platform\, and store the .cpp and .h files in there. That way, when you add various kinds of specialized platforms later on, they'll all be in once convenient and logical place.
- Implement a **factory** for your `Platform` class, so it can be loaded from a `Spawner` description contained in a **.level** file. Do this as follows:

Platform.h

```
namespace ITP485
{
    FACTORY_DECLARE(PlatformFactory, Platform);

    class Platform : public MeshInstance
    {
        GAME_TYPE_DECLARE(ITP485::PlatformFactory);
        // ...

        virtual void init(const Spawner* pSpawner);
    };

    // ...
}
```

Platform.cpp

```
GAME_TYPE_DEFINE(ITP485::Platform, ITP485::MeshInstance,
                 ITP485::PlatformFactory);

void ITP485::Platform::init(const Spawner* pSpawner)
{
    // Init your Platform based on the data in the Spawner.
    // The Spawner contains the description of the platform
    // as provided by the user in the .level file.
}
```

- Add suitable renderable geometry to your platform. For now, you can use the OGRE "prefab" box mesh. To do so, simply call your `Platform`'s `setMesh()` function (which is inherited from `MeshInstance`) with the special mesh name `"Box"`. This will end up initializing your Platform with an `Ogre::SceneManager::PT_CUBE` "prefab" cube mesh. Later you can replace the box meshes with more interesting custom mesh(es).
- Scale and translate the renderable mesh to match the AABB.
- Verify that you can define two or more platforms in the **.level** file, that they load into the game, and that they render properly.
- Use the engine's **debug drawing** system to render the AABB of each platform. Verify that it matches up reasonably well with the platform's visible mesh.
    - To do this, you'll need to `#include "Engine/Render/DebugDraw.h"`, and then you can call `DebugDrawManager::get()` to obtain a reference to the singleton debug-drawing manager.
    - Call its `addBox()` member function to draw each platform's AABB.

7.3.3. Week 8

- Add a singleton `PlatformManager` class. This class will keep track of all `Platform`s in the game world, and allow the `Player` class to ask questions like "is there a platform at such-and-such a location in the world?"

  - If you derive your Platform class from `Ogre::Singleton`, you'll get a nice consistent singleton interface. See for example `GameObjectManager`.

- Whenever a `Platform` is constructed (i.e. in `Platform::init()`), register it with the singleton `PlatformManager`. Likewise, whenever a `Platform` is destroyed (i.e. in `Platform::~Platform()`), unregister it with the singleton manager. You can use a roll-your-own linked list, or an STL `std::list`, to keep track of all the `Platform`s within the manager object.
- Design and write code to support the following API, which will be used by the `Player` as it traverses the game world:

```
class PlatformManager : public Ogre::Singleton<PlatformManager>
{
public:
    // Given a ray from point 'start' to point 'end', this function should
    // return the first Platform struck by the ray, or NULL if none. The
    // Player will call this every frame, with a downward ray, in order to
    // find a new platform during free-fall.
    Platform* rayCast(const Ogre::Vector3& start,
                      const Ogre::Vector3& end);

    // ...
};

class Platform : public MeshInstance
{
public:
    // This function should return the walkable plane of the platform.
    // The Player will call this to obtain its walking surface when on
    // a platform. Assuming platforms are represented as AABBs, this
    // will always be a horizontal plane with an upward-facing normal.
    Ogre::Plane getWalkablePlane() const;

    // This function should return true if the given point lies on the top
    // surface of the platform (to within some reasonable tolerance), and
    // is within the platform's boundary. The Player will call this every
    // frame to determine whether it has walked off the edge of its
    // current platform.
    bool isPointOnPlatform(const Ogre::Vector3& point);

    // ...
};
```

- Work with the Player team to test and integrate your API. The Player should be able to walk on, fall off, and land on platforms.

7.3.4. Week 9

- Complete the work you started last week, and test the player's new "platforming" mechanics.
- Work with the Player team to fine-tune the mechanics until they "feel" good.

- Design one or more test levels with platforms laid out to test every "edge case" you can think of. Here are some ideas:
  - Immediately adjacent platforms (i.e. the *y* coordinates of the walkable surfaces of the two platforms are identical, and the platforms are *just* touching along the *x* or *z* axis). The Player should be able to walk over such "seams" without falling through.
  - Adjacent platforms with a gap – how large of a gap can the Player "make," given its jump mechanics?
  - Small navigable "curbs." The Player should be able to walk up small curbs without the human player pressing the JUMP button. How large of a curb do you want to allow?
  - Walls. When the Player encounters a platform that is higher than its current platform but too high to walk up like a "curb," the Player character should stop dead in its tracks. (NOTE: You'll need to add some more collision-detection code to make this work properly.)

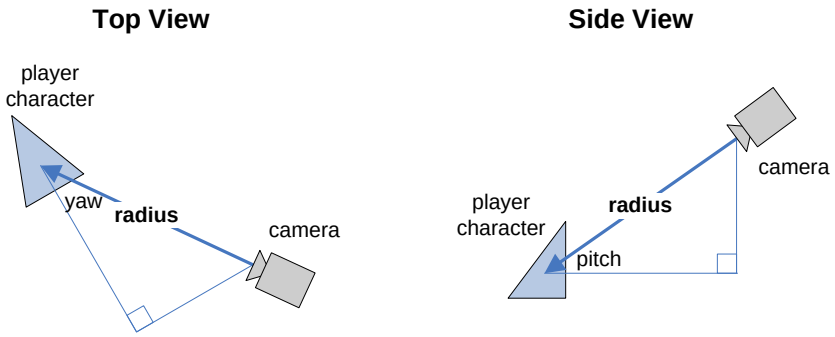
7.3.5. Weeks 10–14

- Once the Player team is satisfied with the basic "platforming" mechanics you've helped implement, the Environment team can turn its attention to designing and implementing some advanced platform types. Here are some ideas:
  - **Moving** platforms. Perhaps a platform that can slide back and forth along a line segment in the *xz* plane. Or elevators that slide up and down along a vertical line segment. Or a platform that remains axis-aligned while sliding along an arbitrarily-oriented line segment. Or a platform that follows a spline… In what ways could the motion of a platform be triggered?
  - **Disappearing/reappearing** platforms.
  - Platforms that **scale** up and/or down. Perhaps each platform starts scaling down to nothingness when first touched, forcing the player to jump off quickly.
  - **Slippery** platforms. **Electrified** platforms that damage or kill the player. What other kinds of **surface properties** might your platforms have? How about changing the surface properties over time?
  - **Rotating** platforms – this will necessitate converting your collision representation from axis-aligned boxes (AABB) to oriented boxes (OBB).

7.4. Camera Team

7.4.1. Overview

- The Camera team's primary job is to implement a "follow cam" that keeps the player on-screen in a pleasing way while accentuating the movements of the player character, and not interfering with the human player's ability to see and control his or her character.
- One possible design models the camera's position and orientation via **cylindrical coordinates** (yaw, height, radius) or **spherical coordinates** (yaw, pitch, radius), with the player character at the origin.



- The camera should be on **springs**, so that if the player character suddenly changes direction or speed, the camera will "lag" behind a little bit. This helps to accentuate the movements of the player character, and also makes the camera "feel" less rigid.
- The human player should also be able to rotate the follow cam manually about the player in order to get a desired viewing angle.
- Because the game supports split-screen multiplayer, up to two "follow cams" may be active simultaneously, each one rendering to a different viewport on-screen.

7.4.2. Week 7

- Start by having a look at the `DebugCamera` class, its base class `Camera`, and the `CameraManager` singleton.
  - A `Camera` object in our engine is just a wrapper around the OGRE camera system.
  - The `CameraManager` does most of the heavy lifting, setting up the OGRE camera(s) and managing switching between cameras.
  - The `Camera` class has a virtual **`update()`** function, whose job it is to update the position and orientation of the underlying OGRE camera every frame.
- A `FollowCamera` class has already been provided for you, but right now it is just a *copy* of the `DebugCamera` class. Your job will be to turn this into a true "follow cam" for the game.

- You can switch back and forth between the DebugCamera and the FollowCamera by hitting the 1 key or the HOME key on the keyboard. Try running the game in split-screen multiplayer mode by pass "/2" on the command line. Notice that there are now two FollowCamera instances, but still only one DebugCamera.
- Start by making the FollowCamera follow the player character around in a rigid fashion. Establish the **ideal** cylindrical coordinates of the camera relative to the player – how far back? how high above? use a zero yaw, i.e. the camera should always be directly behind the player character, facing in the same direction as the player, to start with.
- Implement your virtual `update()` function to achieve this rigid follow behavior.

7.4.3. Week 8

- Add **springs** to your follow cam.
- The engine already provides a template class called `Spring` which implements a **critically-damped spring**. This is technically a spring coupled with a viscous damper, and tuned such that the apparatus returns to equilibrium as quickly as possible without oscillating. A critically-damped spring-damper system is a great way to provide a nice looking "ease in" effect to a camera or animation.
- To add springs to your camera, you should maintain two sets of cylindrical coordinates – the ideal or goal coordinates, and the current coordinates. If the player is not moving, the goal and current coordinates will match, and the damped spring will be in equilibrium. But if the player suddenly changes position, orientation or speed, the current will diverge from the goal (which follows the player rigidly), and the damped spring will cause the current to "catch up" to the goal over a brief period of time.
- Consider whether you want your camera to always automatically swing back behind the player, or whether you want it to stay put when the player stops moving.

7.4.4. Week 9

- Add the ability for the human player to adjust the camera's pitch and yaw manually.
- How can you reconcile the camera's natural rotation (driven by the springs and your definition of the "ideal" camera position and orientation) with the human player's manual override of the pitch and yaw?
- Camera controls are best provided via the left joypad stick, although you may want to provide a "mouse look" feature and/or keyboard controls as well.

7.4.5. Weeks 10–14

- Test the behavior of the camera while putting the player character through literally *all* of its paces. How does the camera fare when the player turns abruptly? accelerates rapidly? stops quickly? falls? dies? is attacked by enemies?

- Fine-tune the camera under every imaginable scenario to ensure that the human player's experience is optimal. Ask yourself: Where would the human player *want* to be looking right now? Does my follow cam generally provide the best vantage point in most situations? If not, how can you tweak the logic of the camera to achieve a better result?
- You may also want to implement specialized cameras, time permitting. Here are some ideas:
  - An "intro" camera that flies through the game world on a spline, showing the player what to expect before gameplay begins.
  - A "victory" camera for the winner.
  - Special "death cams" for when the player falls to his death, or is killed by enemies, etc.
  - Special-case camera(s) for tight spots. For example, if the player enters a long enclosed tube or hallway, perhaps the camera goes into a mode where it aligns itself with the passageway and cannot be manually rotated until the player emerges from the other side.
  - Camera collision. Does your camera sometimes clip through visible objects? If so, can you find a way to detect these collisions and move the camera out of the way? Note that it's usually best not to rotate the camera, as this can disorient the player. (Remember, player controls are camera-relative.) It's often better to zoom in than to rotate to avoid a collision.

7.5. Hazards/Enemies Team

7.5.1. Overview

- The Hazards/Enemies Team is responsible for designing and implementing one or more obstacles to get in the way of the player achieving his or her goals in the game.
- The game design is entirely open-ended and up to you. Here are a few examples of the kinds of things you might want to implement:
  - "Hazard" platforms—electrified platforms, fire platforms, trap door platforms, etc. (Work with Environment team to design and implement these.)
  - Hazards that block the player's path—bursts of fire or hot steam, blades or spikes that pop out at periodic intervals, etc.
  - Walls that close in, slam down, or push the player off its platform periodically.
  - Simple path-based enemies that follow a predetermined patrol route—player must time traversal to avoid them.
  - Characters or turrets that shoot lasers or other projectiles at the player character as it passes by.
  - AI-controlled characters (non-player characters, i.e. NPCs) that chase and attack the player.
- Hazards might be dangerous all the time, or only periodically. They might be triggered by the player, or they might activate automatically, either when the player gets close enough, or periodically. Can the player disable the hazard in some way? Or is the hazard impervious to the player?
- You may also want to design and implement some treasures and/or power-ups for the player to collect.

7.5.2. Week 7

- Brainstorm a big list of all the kinds of hazards/objects you might want in your game.
- Select a small number of your best ideas for detailed design and implementation.
  - The number of hazard types you select should depend on how ambitious the ideas are. If you have very ambitious ideas, select only one or two of them for initial implementation. If the ideas seem like they will be pretty simple to implement, then feel free to tackle three or four ideas at first.
  - Be careful not to bite off more than you can chew. Document *all* of your ideas, and plan to come back and implement more of them if (and only if) time permits.
- Design the mechanics of your top one to four hazard types. How will they work? How will they be triggered? How will they affect the player character? What kinds of visual and audio effects will be required for each?
- Start prototyping *one* of your hazard types.

7.5.3. Weeks 8–14

- Prototype, complete and fine-tune each of your hazard types.
- Work with Environment, Player, Camera and Effects teams to ensure that your hazards integrate properly with the rest of the game.
- Once you have your initial hazard types fully implemented, you can go back and implement more of your original ideas, time permitting.

7.6. Effects Team

- The charter of the effects and lighting team is to add visual and audio polish to the game. While the rest of the team is focused on functionality and player/enemy mechanics, your job is to focus on aesthetics.
- What you do to beautify and polish the game is up to you. Here are the technologies you will probably end up working with:
  - **Lighting.** Learn OGRE's lighting system, and the Light class in the ITP485 engine (derived from GameObject). Add lights to the scene to make it look the best it can. Experiment with OGRE's various shadow techniques. Be sure to balance image quality against performance—obviously you mustn't add so many lights and shadows that the frame rate grinds to a halt.
  - **Materials and Shaders.** Learn OGRE's material system, and use it to make the 3D meshes in the game look the best they can. If you're ambitious, you might even try your hand at writing a custom shader or two!
  - **Particle Effects.** Learn OGRE's particle system, and use it to add ambient particle effects, effects on platforms (e.g. fire platforms, electrified platforms), effects for hazards (e.g. steam blasts, lasers, poisonous gas), dust puffs on the characters' feet, and so on.
  - **Heads Up Display and Game GUI.** Learn the OGRE overlay system; you will use it later in the project to implement the game's heads up display (HUD) and wrapper screen(s). (Wrappers might include a title screen, game options menu if any, and screens or HUD elements to tell the player when he or she has won or lost the game.)
  - **Sky Box.** OGRE provides some simple facilities for rendering a sky either as a box or hemispherical map. Research how this is done so you can add a sky to your game world if one is needed.
  - **Audio.** Learn the Microsoft XACT audio authoring tool and runtime API. Take a look at `Sound.h` and `Sound.cpp` in the ITP485 engine code—much of the groundwork has already been done for you. Use XACT to add background music and audio effects to the scene.

7.6.1. Weeks 7–9

- You will most likely spend the first three weeks learning the technologies you will need to work with later in the project.
- Divide and conquer—if there are two of you, split up the technologies. Perhaps one of you can be the rendering expert and the other the audio expert, for example.
- Learn about lighting, materials and particle effects in OGRE, by reading the OGRE documentation and wiki, reading the OGRE sample code, and experimenting with the foundational code provided to you by the ITP485 engine. Try to implement some simple lighting, material and/or particle effects—there's no better way to learn than to do.

- Learn about OGRE's overlay system for 2D graphics. Experiment with placing some simple heads up display elements on-screen. These needn't be final game assets—you're just experimenting at this point.
- Learn about Microsoft XACT toolkit and runtime API. Skim the XACT documentation on MSDN. Look through the foundational code in Sound.cpp in the ITP485 engine. Playing audio clips is not particularly difficult, but to really fine-tune your game's audio can be an involved task, so it's best to understand what XACT is capable of up-front. As with visual effects, it's probably best to experiment by actually trying to play some music and sound-effects. They don't have to be the final effects needed by the game, just some samples to get your feet wet.

7.6.2. Weeks 10–14

- Once you have a reasonably good understanding of how OGRE's and XACT's technologies work, you will be ready to produce final effects for the game.
- By Week 10, enough of the game mechanics and world layout should be implemented that you will know what exactly is needed by way of visual and audio effects.
- The game world(s) should be laid out by Week 10 or 11, so you should be able to light the world at this point.
- Work closely with the other teams to ensure that the final game will look and sound its best, and that all necessary HUD elements are functioning properly.