Physics Wrapper

Getting Started with a Demo Program

Matthew Langford

Updated: 2017 - 04 - 02

Contents

1 Setting up a development environment

If you already have way of writing and compiling OpenGL programs, and you feel confident getting Bullet Physics compiled and linked, then you can skip this part.

Otherwise, we're going to set up a development environment using MSYS2 and Code::Blocks. We assume that you are using a modern 64-bit system, so we will be downloading the 64-bit versions of each program.

1.1 Setting up MSYS2

- Download `msys2-x86_64-20161025.exe` from `http://www.msys2.org/`.
- Follow the steps on the website to get MSYS2 set up
  - Run `msys2-x86_64-20161025.exe` and set the install directory etc.
  - Run msys2 (should be on the start menu now) to get an empty shell.
  - In the window, enter the command `pacman -Sy pacman` to update the package manager.
  - Close the window.

1.2 Installing the dev tools

- Restart msys2.
- Install the C++ compiler
  - `pacman -S mingw-w64-x86_64-gcc`
- Make sure that the following locations have been added to your system path.
  - `<install>\mingw64\bin`
  - `<install>\usr\bin`
  - NB: You can access the path from `Control Panel > System > Advanced System Settings > (TAB) Advanced > Environment Variables`. (See Figure 1).
- Two more packages to install next:

  - `pacman -S --noconfirm mingw-w64-x86_64-gdb`
  - `pacman -S --noconfirm mingw-w64-x86_64-make`
- Find `mingw32-make.exe` and rename it `make.exe`.
  - I found it in `<install>\mingw64\bin`
- Download Premake 4.x from `https://premake.github.io/download.html#v4` and put the executable into `<install>\usr\bin`.
- Finally, we need OpenGL and Bullet Physics stuff:
  - `pacman -S --noconfirm mingw-w64-x86_64-glfw`
  - `pacman -S --noconfirm mingw-w64-x86_64-glm`
  - `pacman -S --noconfirm mingw-w64-x86_64-glew`
  - `pacman -S --noconfirm mingw-w64-x86_64-bullet`

1.3 Getting an IDE

We can compile the physics applications from msys2 using gcc, but it's generally easier to have an IDE or makefile to keep track of everything for you. If you don't already have a preference, I recommend Code::Blocks, available to download at `http://www.codeblocks.org/downloads`.

Once Code::Blocks is installed, you need to tell it how to use the tools we just installed. Go to `Settings > Compiler` and create a new compiler by copying the default one. Name it something like *MSYS GNU GCC*. Go to the `Toolchain executables` tab and fill in the fields as in Figure 2.

Set this compiler as default afterwards.

2 Downloading the Physics Wrapper

Get the Physics Wrapper code from `https://github.com/Mattadon/ML-Bullet-Wrapper` if you haven't already done so. Clone/extract the files to a memorable location.

The provided version of the library in `lib/` should work fine with Microsoft and MinGW compilers. Otherwise, you can compile the library for yourself. There is a provided `premake4.lua` script to generate project files from the provided sources.

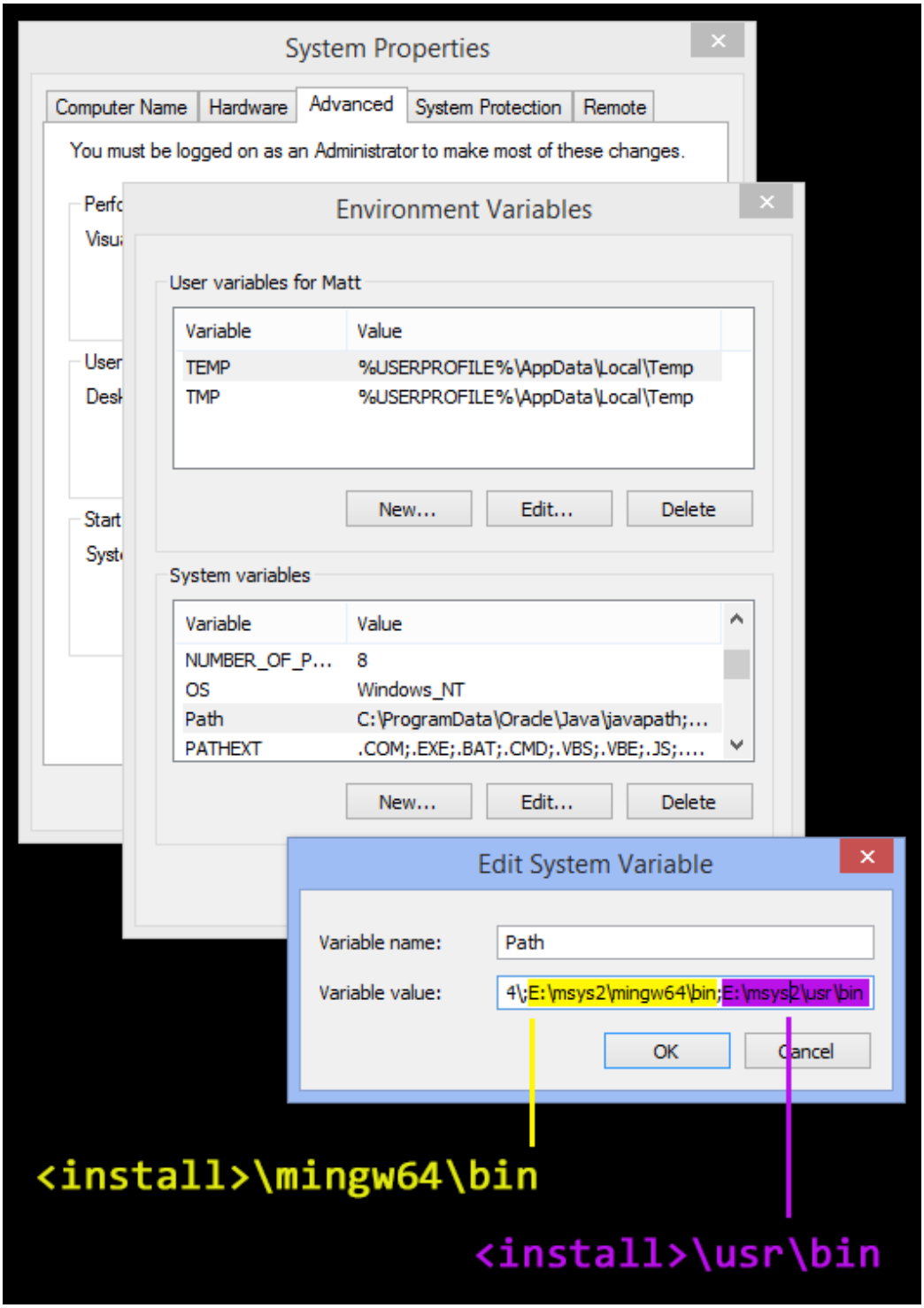


Figure 1: Checking the path variable. In this example, I have installed msys2 to E:\

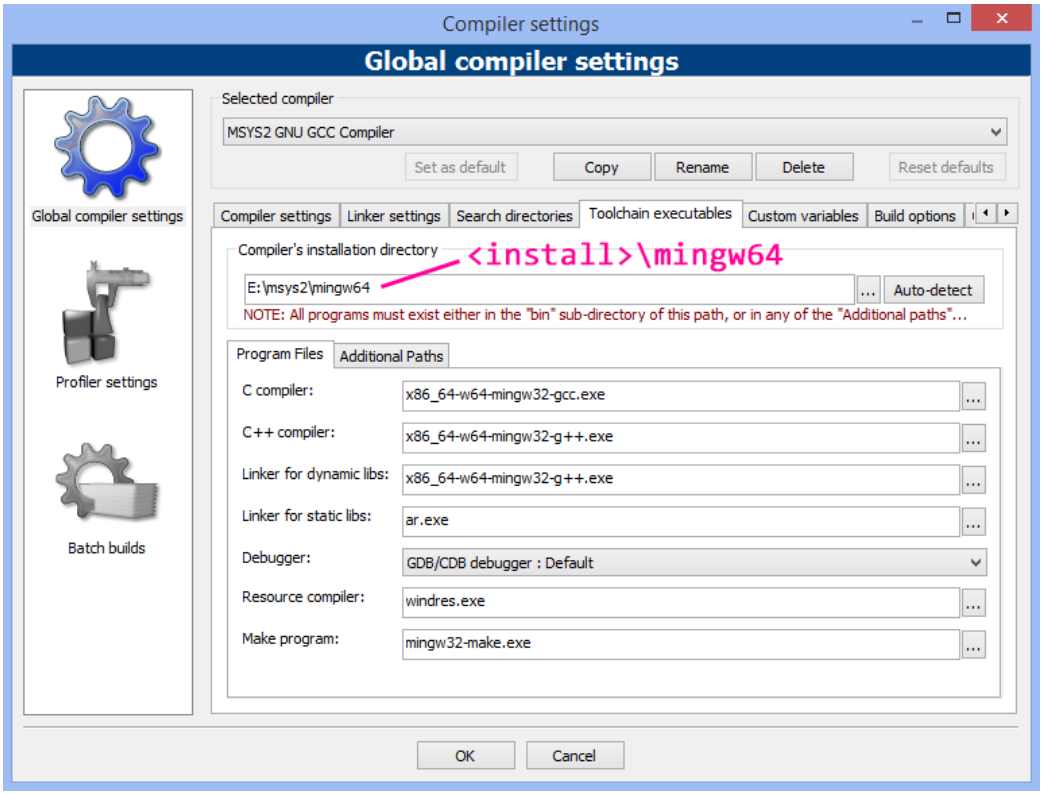


Figure 2: Setting up the Code::Blocks compiler. In this example, I have installed msys2 to E:\

2.1 (Optional) Generating the library from source

This assumes that you have set up your development environment by following the steps in Section 1.

- Open msys2 and navigate to the location of the Wrapper (familiarise yourself with the `cd` and `ls` commands to do this).
- Run `premake4 codeblocks` to generate a Code::Blocks project file.
- Open the project by double-clicking on the `.workspace` file that is produced.
- You may have to remind Code::Blocks to use your compiler.
  - Right-click on the project root in the 'Management' window and select 'Build options'.
  - Check that your *MSYS2 GNU GCC* compiler is selected.

- Under the 'Linker settings' tab, check that `libBulletDynamics`, `libBulletCollision` and `libLinearMath` are linked in that order.
- Under the 'Search directories >Compiler' tab, add a new item for the location of your Bullet Physics include-files. By default, I have it at `<msys2 install>\mingw64\include\bullet`.
- The program should now compile and update the library file in `lib/`.

3 Setting up the Demo Application

You can get the demo application at:

`https://github.com/Mattadon/ML-Bullet-Wrapper-Tutorial`.

When you want to make an application using the Physics Wrapper, the things you have to worry about are:

- Linking to the static library in `<Wrapper location>/lib/libBulletWrapper.a`.
- Linking to Bullet Physics libraries to provide functionality for the wrapper.
- Including the contents of the `<Wrapper location>/SimpleBulletWrapper/include/` directory.

If you know how to do this - great! If not, the following section walks through the steps to get the demo application running using the environment we set up in Section 1.

3.1 Using the demo with MSYS2 and Code::Blocks

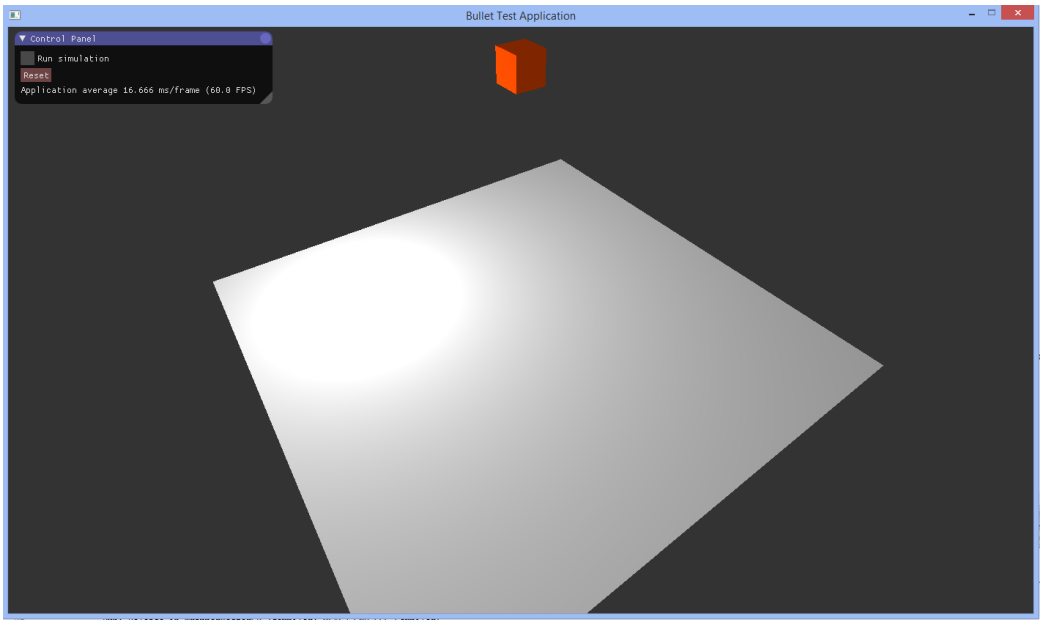
- Open msys2 and navigate to the location of the demo application code (familiarise yourself with the `cd` and `ls` commands to do this).
- Run `premake4 codeblocks` to generate a Code::Blocks project file.
- Open the project by double-clicking on the `.workspace` file that is produced.
- You may have to remind Code::Blocks to use your compiler.
  - Right-click on the project root in the 'Management' window and select 'Build options'.

- Check that your *MSYS2 GNU GCC* compiler is selected.
- Under the 'Linker settings' tab, check that the following libraries have been linked correctly, in order:
  * `glew32`
  * `glfw3`
  * `opengl32` (These three are for OpenGL to draw stuff for us!)
  * `libPhysicsWrapper` (This is the wrapper library!)
  * `libBulletDynamics`
  * `libBulletCollision`
  * `libLinearMath` (These three supply the bullet code that underlies the wrapper.)
  * `gdi32` (This is required by some non-Microsoft compilers for the image-loading component of the wrapper, based on CImg[1].)
- Under the 'Search directories >Compiler' tab, add:
  * A new entry for the location of the Physics Wrapper header code. In a serious environment, you would probably add this to an overall `Include` location, but for this demo, we'll just just look directly in the wrapper source code. Add an entry for `<Wrapper location>\SimpleBulletWrapper\include`.
  * A new entry for the location of your Bullet Physics include-files. By default, I have it at `<msys2 install>\mingw64\include\bullet`.
- Under the 'Search directories >Linker' tab, we will tell the compiler where to find the wrapper library itself. Again, you would normally add the library to a shared `Lib` location, but since we're just using the one, add an entry to `<Wrapper location>\lib`.
- The program should now compile and run! (Hopefully)

[1] `http://cimg.eu/`

4 Using the program

If the program has compiled correctly, you should see the following when you load up the program:



You can control the camera using the mouse.

- Scroll Wheel: Zoom in and out
- Middle-click + drag: Rotate the camera
- Middle-click + Shift + drag: Pan the camera

In this simulation, a cube is dropped onto an infinite plane (represented here by a finite plane - pretend it extends in all directions). The Space-bar and 'Run simulation' toggle in the UI will start/stop the simulation. The 'Reset' button in the UI will return all objects to their initial positions.

5 Understanding the program

The interesting Physics Wrapper stuff is happening in Source.cpp, so open that in Code::Blocks (or whatever editor you are using).

Figure 3 acts as a very simple example of what you can do using the Physics Wrapper. Think of it as our equivalent of a 'hello world' application.

```
//=========================//
//Set up the physics engine//
//=========================//

//Set the world to have 1x gravity and use degrees
PhysicsWorld* world = new PhysicsWorld(1.0f, false);

//========================//
//Add some physics objects//
//========================//

//Add your objects to this vector to get them drawn and reset
std::vector<GameObject> physicsObjects;

//Create a unit cube directly above the origin of the world
PhysicsBox defaultBox(world);
Mesh boxMesh(getBoxVertices(1.0f, 1.0f, 1.0f), getBoxIndices(), glm::vec4(1.0f, 0.3f, 0.0f, 1.0f)
    );
GameObject boxObject(&boxMesh , &defaultBox , glm::vec3(0.0f));
physicsObjects.push_back(boxObject);

//Create an infinite plane in the XZ plane, passing through the world origin
PhysicsPlane defaultPlane(world);
Mesh planeMesh(getXZPlaneVertices(20.0f), getPlaneIndices(), glm::vec4(0.8f, 0.8f, 0.8f, 1.0f));
GameObject planeObject(&planeMesh, &defaultPlane, glm::vec3(0.0f));
physicsObjects.push_back(planeObject);

//===============//
//Simulation loop//
//===============//

/*Game loop keeps running so that the window doesn't just quit.*/
while (!glfwWindowShouldClose(window))
{
    //Does openGL stuff at the start of the loop
    updateApp(&physicsObjects, world);

    //================//
    // Run simulation //
    //================//

    if(simulationRunning)
        world->stepWorld(deltaTime);

    //Does openGL stuff at the end of the loop
    lateUpdateApp(&physicsObjects, world);
}

//========================//
//Clean up after ourselves//
//========================//

delete world;
```

Figure 3: Extract from `Source.cpp` in our Demo program.

Most of the physics simulations that you set up using this wrapper will follow the structure that we have used here:

- Set up the world
- Add the objects that we want at the start of the simulation
- In the main loop, update the world

5.1 Setting up the world

The first thing we do when making a physics simulation is to create a world for the physics objects to inhabit.

Some physics engines let you set all sorts of options and algorithms to support the running of a simulation, but we have kept it as simple as possible: when you set up a world, all you need is the gravity and whether you are going to be using radians for your angles.

We have put the PhysicsWorld on the heap using the `new` keyword, so we have to make sure to `delete` it at the end of the program.

See Section 4 of the user guide included in the library download for usage details of the PhysicsWorld.

5.2 Adding physics objects

Next, before we enter the main game loop, we set up the physics objects that we want to exist at the start of the simulation.

The Physics Wrapper has support for rigid bodies, which means that our physics objects will not deform over time.

In this very simple scenario, we add just a cube and an infinite plane. We use some special constructors that don't take any arguments to set up initial conditions - we just pass in the pointer to the PhysicsWorld. These constructors automatically set up the cube directly above the plane. Though this obviously makes the resulting simulation rather unexciting, you may find these useful for when you are just trying to get a new project working for the first time.

You can find the full list of supported rigid bodies in the user guide.

The three lines that come after the physics object definition (`PhysicsBox defaultBox(world)`) are not part of the wrapper, but are used by the demo program to draw the objects that we create using our wrapper:

`Mesh boxMesh(...)` Creates a mesh that will represent our object. Meshes are the 3D model that we use to draw the shape on the screen. Meshes take three arguments:

- A vector of vertices
- A vector of vertex indices
- A colour, represented by a RGBA vector (If you're not sure about how to use RGB colours, there's a helpful tool here: `https://www.w3schools.com/colors/colors_rgb.asp`. The A component should always be 1.0f in this program).

The demo application comes with some support methods to generate the vertices and indices for certain shapes - there is a full list in Section 6.

`GameObject boxObject(...)` essentially just wraps the physics object and the mesh up in a nice simple structure. This will take care of moving the mesh to the position and rotation of the physics object for you. Its arguments are:

- A pointer to the mesh to use
- A pointer to the object you want to use
- A vector that applies a permanent offset between the mesh's and physics object's origins. For most objects, this should just be 0 for all directions.

  However, heightmaps require you to apply an offset of `glm::vec3(0.0f, -maxHeight / 2, 0.0f))` in order for the collisions to line up with the mesh.

  Additionally, this is useful for planes that need to be drawn away from the world origin, as the underlying implementation for planes means that the mesh won't get automatically moved to the correct position like in other objects.

`physicsObjects.push_back(...)` adds the object that you've created to a vector of game objects. The program will use this vector to draw these objects, and to reset them when you hit the reset button!

5.3 Updating the world

If left to its own devices, the world will just stay static, not doing anything. We need to tell the world that some time has passed, and that it needs to move its objects to reflect the passage of time!

`world->stepWorld(deltaTime)` tells the physics world to move `deltaTime` into the future. Obviously, if we want the simulation to look smooth, deltaTime should be a small value so that objects only change position by a small amount between steps. In the demo application, deltaTime is automatically generated as the time since the last repetition of the main loop. But doing it this way, we can make the speed of the simulation independent from the framerate of our application! This means that people with really good computers won't see weird sped-up simulations because they can run the program really fast!

6 Meshes supported by the demo

Boxes

```
//Boxes with non-equal side length
static std::vector<struct Vertex> getBoxVertices(float width, float height, float depth);
static std::vector<GLuint> getBoxIndices();
```

Use these functions when making PhysicsBoxes. The sides are in the same order for both the physics object and the mesh.

Planes

```
//Planes
static std::vector<struct Vertex> getXYPlaneVertices(float sideLength);
static std::vector<struct Vertex> getXZPlaneVertices(float sideLength)
static std::vector<GLuint> getPlaneIndices();
```

Use these functions with a horizontal or vertical PhysicsPlanes. In this version of the demo application, you can't create other types of plane - and the way that PhysicsPlanes are made means that it won't get rotated to the correct position in space when the application tries to draw it. Until this is patched, you can substitute with large, thin boxes. (Sorry)

Terrains (heightmaps)

```
//Terrains
static std::vector<struct Vertex> getTerrainVertices(   std::vector<float>* data,
                                                        int dataDepth,
                                                        int dataWidth,
                                                        float worldDepth,
                                                        float worldWidth);
static std::vector<GLuint> getTerrainIndices(int dataDepth, int dataWidth);
```

Use these functions with a PhysicsHeightmap.

`getTerrainVertices` requires:

- Vector of floats, containing the heights above the lowest level of each point on the heightmap.
- The number of points in the x direction
- The number of points in the z direction
- The length of the side in the x direction
- The length of the side in the z direction

`getTerrainIndices` only needs the arguments for number of points in the x and z directions. These fields are easy to fill out if you've previously loaded a PhysicsHeightmap (and why else would you use this?):

```
//Create a heightmap physics object
PhysicsHeightmap heightmap("image/pyramid.bmp", 30, 15, maxHeight, glm::vec3(0, 0, 0), world);
//Get the height data out of the physics object
HeightfieldData heightData = heightmap.getHeightMapData();

//Use the height data to create a mesh
std::vector<struct Vertex> verts = getTerrainVertices(heightData.getData(), heightData.getWidth()
      , heightData.getDepth(), 30.0, 15.0);
std::vector<GLuint> inds = getTerrainIndices(heightData.getWidth(), heightData.getDepth());

Mesh terrainMesh(verts, inds, glm::vec4(0.5, 0.5, 0.5, 1.0f));
```

`HeightfieldData` will keep track of the most of the stuff you need to plug into the mesh constructor, so make sure to get a copy of it from the heightmap you want to draw.

Spheres

```
//Spheres
static std::vector<struct Vertex> GetSphereVertices(float radius);
static std::vector<GLuint> GetSphereIndices()
```

Use these with PhysicsBalls.

Cones

```
//Cones
static std::vector<struct Vertex> getConeVertices(float height, float radius);
static std::vector<GLuint> getConeIndices();
```

Use these functions with PhysicsCones. The axis of rotation is in the Y direction - the same as when creating a PhysicsCone, so the mesh will line up with the body nicely.

Cylinders

```
    //Cylinders with oval cross-sections
    static std::vector<struct Vertex> getCylinderVertices(float depth, float height, float width);
    static std::vector<GLuint> getCylinderIndices();
```

Use these functions with PhysicsCylinders. The arguments are in the same order as in the PhysicsCylinder constructors, and the axis of symmetry is the same too - copy the numbers across and the mesh should line up nicely with the physics object.

In the current version, these look a little weird as some of the faces are missing. Until we work out a fix for this, please just bear with the ugly look. (Sorry again)

Complex meshes

```
    //Meshes from .obj files
    static std::vector<struct Vertex> GetMeshVertices(std::string filename);
    static std::vector<GLuint> GetMeshIndices(std::string filename);
```

Use these functions with PhysicsConvexMeshes and PhysicsConcaveMeshes. Just make sure that you pass these functions the same filename as you did when creating you mesh physics objects.