ASSIGNMENT 3

NEW STARTING CODE

There is a new set of starting code for assignment 3. It has been updated to draw textures and 3D mesh objects.

The only changes to a1.c are some additional extern declarations and some new example code in the -testworld.

You will need to transfer the code you wrote for A1 and A2 from the old a1.c to the new a1.c file.

If you modified graphics.c or visible.c then you will have to transfer those changes to the new versions of those files.

All of the code you wrote for assignments 1 and 2 should work the same. Unpack the new code, move your code for A1 and A2 to the a1.c file, compile it, and it should work exactly the same.

ASSIGNMENT 3 GOALS

1. Timing
2. Two Dimensional Map
3. Adding Textures to Cubes
4. Animated Mesh Movement
5. Visibility Testing for Object

TIMING

The update() function is used to control animation, such as the clouds from A2, and the artificial intelligence for the enemies in the game. It is where you control the game.

It runs when the system isn't refreshing the display. It does not run a predictable intervals. It will run more frequently on faster computers and less often on slower computers.

If you perform animations every time the update() function runs then the speed of the animation wont be predictable. To solve this problem you need to write a system that checks how much time has passed each time the update() function is called. Use this to schedule animation events in the game so they will run at the same speed.

When update runs you should calculate how much time has passed since the last animation update, and if enough time has passed then perform the next step in the animation. For the clouds in assignment 2, decide if the clouds should move in this time step or not.

This scheduler will be used for other animations in the game so write it to be flexible. It is probably a good idea to write a trigger system which advances an animation one step each time it is called.

MAPS

Write a map system which shows the maze and the outdoor level from above.

The map of the outdoor level will only show the boundaries of the level and any items that are on that level, such as stairs and the player. If you want to show the elevation of the outdoor level like a topographic map, you can but it isn't required.

The maze level maps will show the rooms, hallways, player, stairs up and down, blocks that players can jump off (from the gravity assignment), and the MOBS (monsters displayed using mesh objects).

These will look a lot like the original Rogue game.

Use the systems 2D graphics commands to overlay the map on the screen. Make the map large enough for it to be easily seen. It should be at least half the size of the window.

MAPS

There are two versions of the map.

The **full map** will show the entire level with all objects marked on it.

The **fog-of-war map** shows only the places that the player has previously visited.

The mapping system has three states that are updated when the m-key is pressed. Use one state to show no map, the second state to show the full map, and the third state to show the fog-of-war map.

MAPS

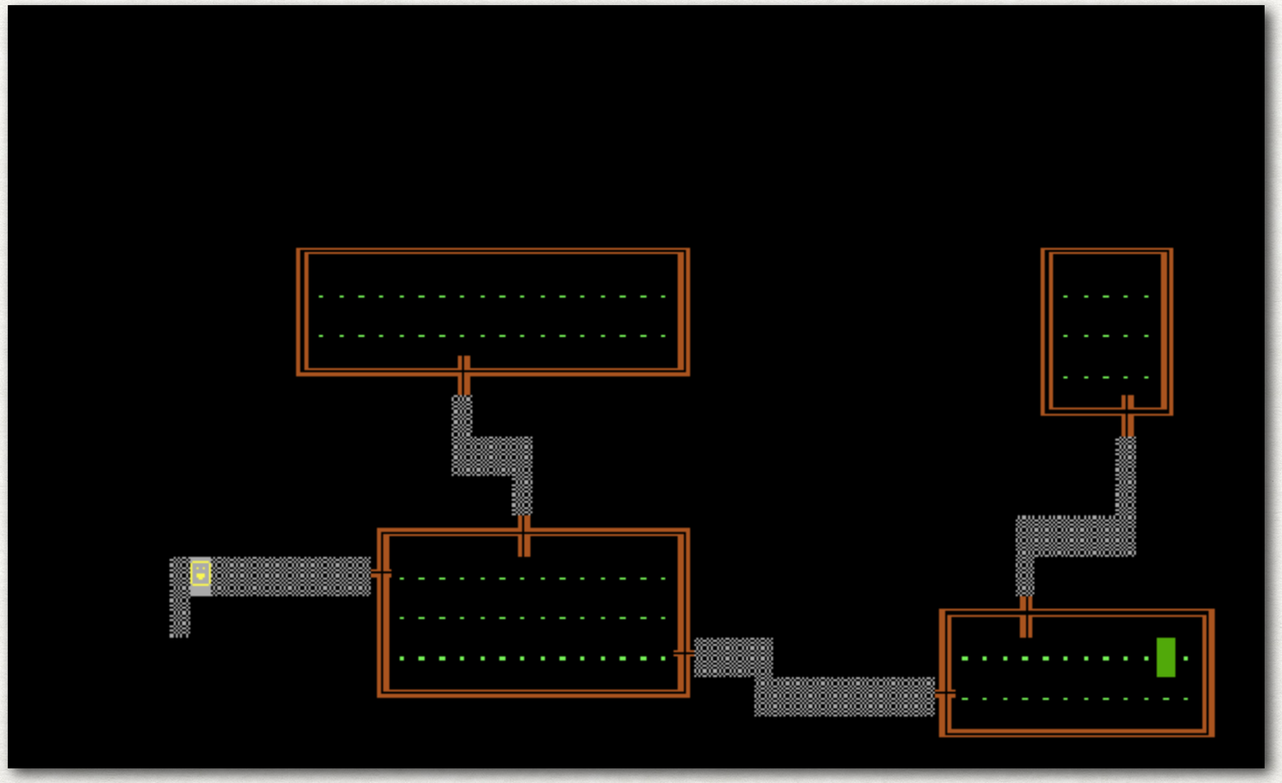
When the player starts a level, the fog-of-war map should only show the room they are in and anything in that room.

MAPS

As they explore they should see more rooms added to the fog-of-war map.

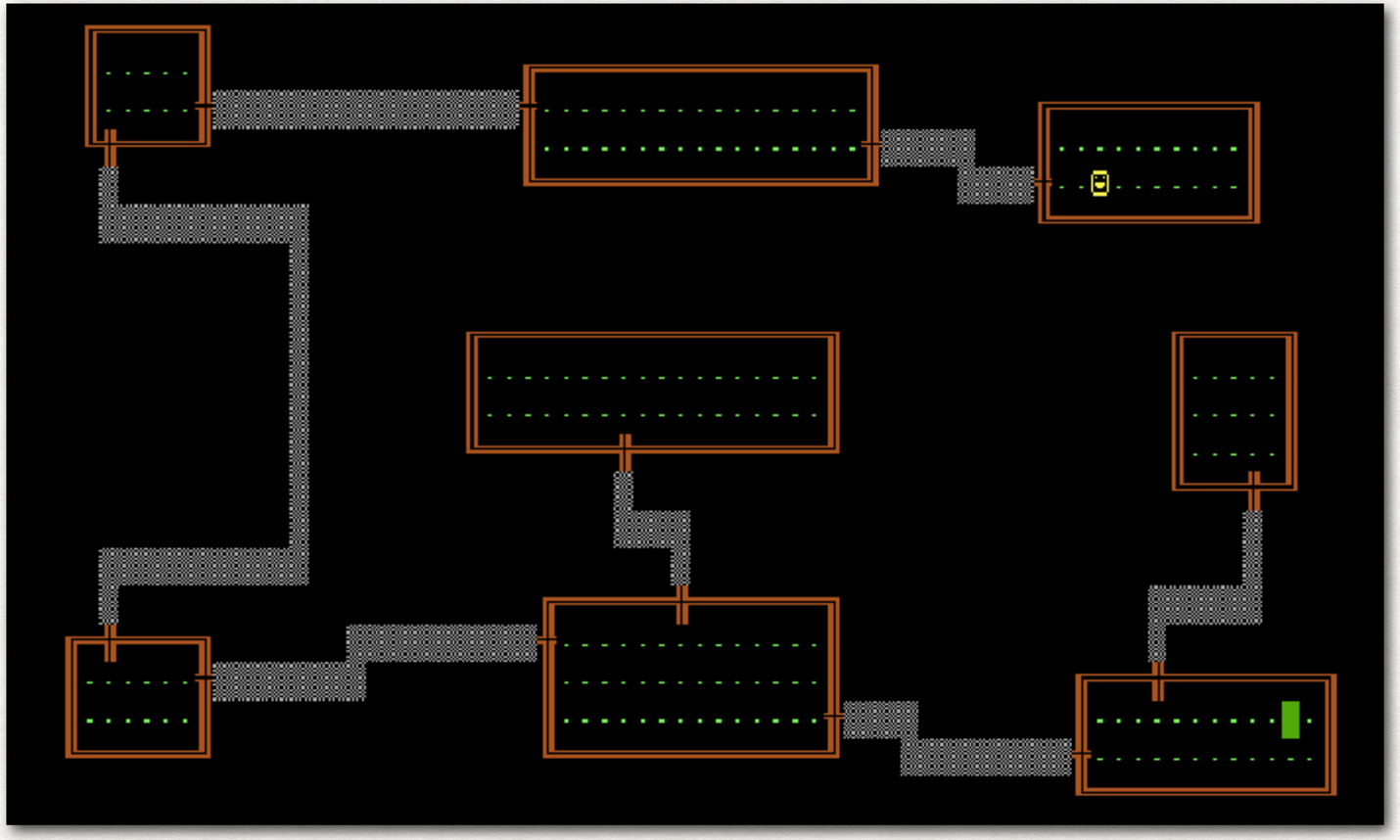
Notice the space in front of the player is visible but not the entire hallway.

The stairs down are the green square on this map.



MAPS

The full map shows all of the rooms, hallways, and items in the rooms. Once the player has visited all rooms and hallways then the fog-of-war map will look the same as the full map.



MAPS

Use the 2D drawing functions.

They must all appear in the draw2D() function. They wont work anywhere else in the program. It is probably a good idea to create a list of items to draw and run through the list in draw2D().

The drawing functions only draw lines and shapes. You can control the transparency of the drawing using the Alpha value of the colour[] array (RGBA).

```
void  set2Dcolour(float colour[]);

void  draw2Dline(int x1, int y1, int x2, int y2, int lineWidth);

void  draw2Dbox(int x1, int y1, int x2, int y2);

void  draw2Dtriangle(int x1, int y1, int x2, int y2, int x3, int y3);
```

TEXTURES

You can apply textures to the cubes now to make the levels more interesting.

There are a set of textures in the /textures directory which you can use.

You can add your own textures. Be sure to follow the formatting instructions in the readme.txt file. Only .ppm files with a resolution of 256x256 will work.

Naming the files correctly is important. The number you give the filename will be the texture number in the game. e.g 10.ppm will be texture 10.

TEXTURES

Textures are added by assigning a texture to a custom colour. The texture will replace the colour on the cube.

Example of adding a texture to a user defined colour id.

```
setUserColour(12, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0);   // create colour id 12
setAssignedTexture(12, 27);       // assign texture 27 to colour 12
world[61][25][50] = 12;           // set the cube at (61,25,50) with colour id 12
```

In this case colour id 12 is set to white (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0).

Texture 27 is assigned to colour id 12.

When the id 12 is placed in the world[][][] array the texture will be drawn on any cubes with the id number 12.

TEXTURES

The user defined colour functions are described in the readme.txt file.

```
setUserColour(12, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0);
```

Ambient RGBA

Diffuse RGBA

To see only the texture colour, set all of the RGBA values to 1.0.

If you want to blend a colour and texture together then you can set the colour to be something other than white. The colour of the cube and the texture will be blended when the cube is drawn.

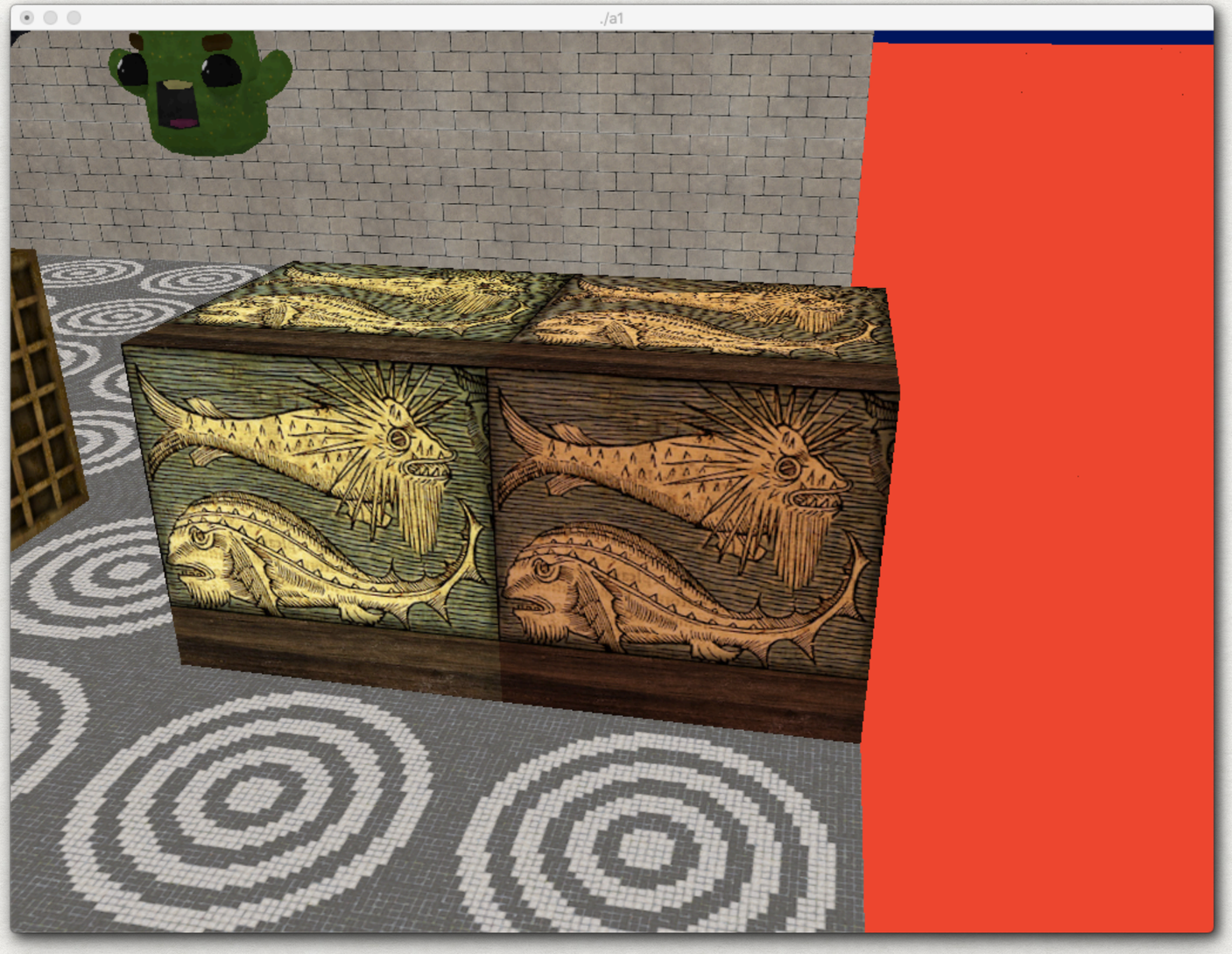
TEXTURES

This shows the same texture but with different RGBA values.

The left cube has RGBA values all set to 1.0. Only the texture colour is visible.

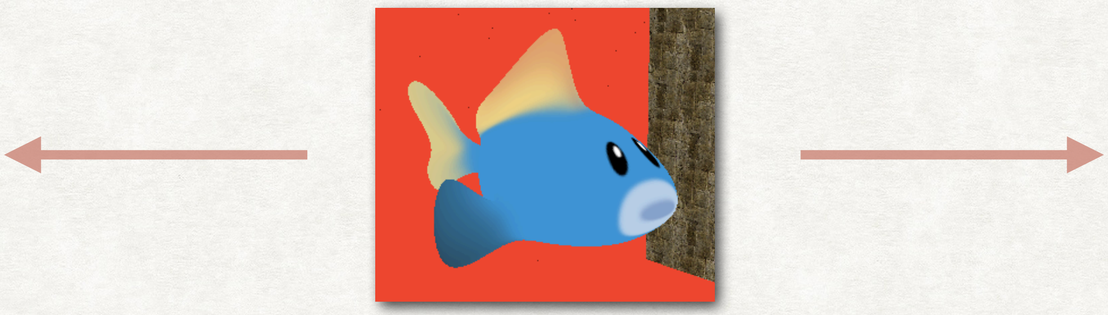
The right cube has the colour values set to:
ambient = (0.7, 0.5, 0.5, 1.0)
diffuse = (0.7, 0.5, 0.5, 1.0)

The result is a reddish colour cube blended with the texture to give a red shaded texture.



ANIMATED MESH MOVEMENT

Place a mesh object in each room. Move the object around the room in a pattern of your choice.

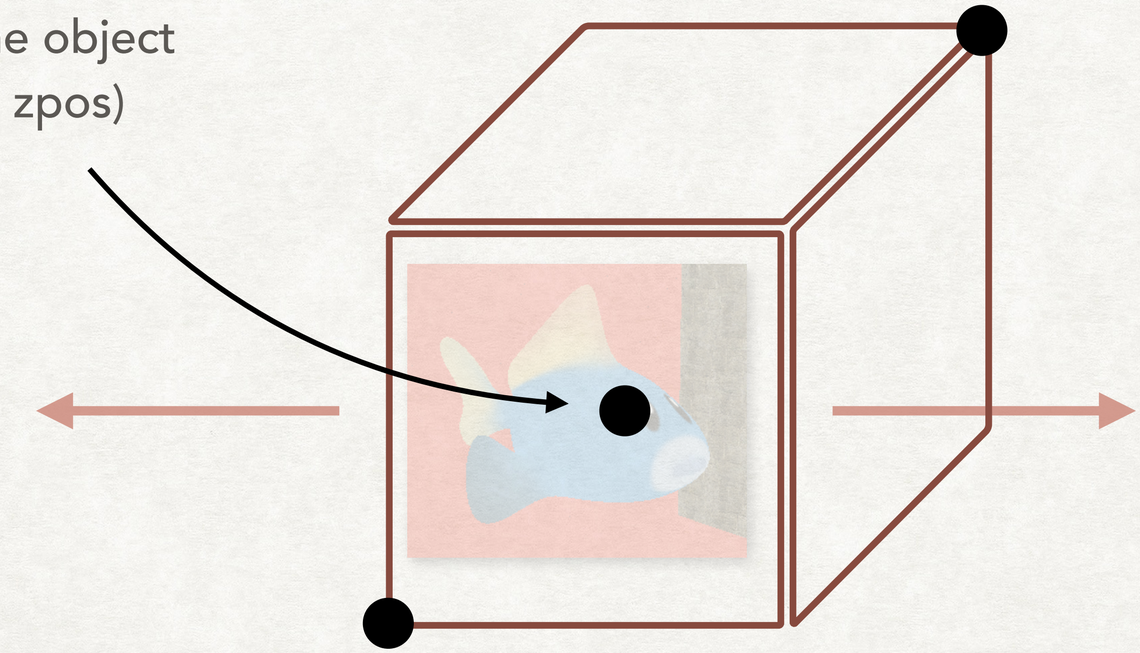


ANIMATED MESH MOVEMENT

The bounding box is going to have to move with the mesh.

(xpos+xMaxOffset, ypos+yMaxOffset, zpos+zMaxOffset)

Centre of the object
(xpos, ypos, zpos)



(xpos-xMinOffset, ypos-yMinOffset, zpos-zMinOffset)

It is probably easiest to store the current location for the mesh (xpos,ypos,zpos) and the minimum and maximum values for the bounding box that are relative to this mesh location. So the minimum corner of the bounding box would be (xpos-xMinOffset, ypos-yMinOffset, zpos-zMinOffset) and the maximum corner would be (xpos+xMaxOffset, ypos+yMaxOffset, zpos+zMaxOffset).

ANIMATED MESH MOVEMENT

There are instructions for controlling the mesh objects in the readme.txt file.

Each mesh has two numbers associate with it. One is a unique id which is used to refer to that instance of the mesh. This is used with all mesh functions to tell the system which mesh you are updating.

The second number is the meshNumber which tells the system which mesh you wish to create. There are four mesh objects included with the system. Their numbers are:

0 = cow
1 = fish
2 = bat
3 = cactus

The meshNumber is only used when the mesh is created using:

```
void setMeshID(int id, int meshNumber, float xpos, float ypos, float zpos)
```

For example:

```
setMeshID(10, 2, 10, 10, 10);
```

creates a mesh with an id of 10, which is a bat (meshNumber = 2) at world location (10,10,10). This can be confusing so it might be worth making some constants for the mesh names.

VISIBILITY TESTING FOR AN OBJECT

We don’t normally want to draw all objects in a world because it requires the graphics system to do a lot of work trying to draw objects that the user cannot see.

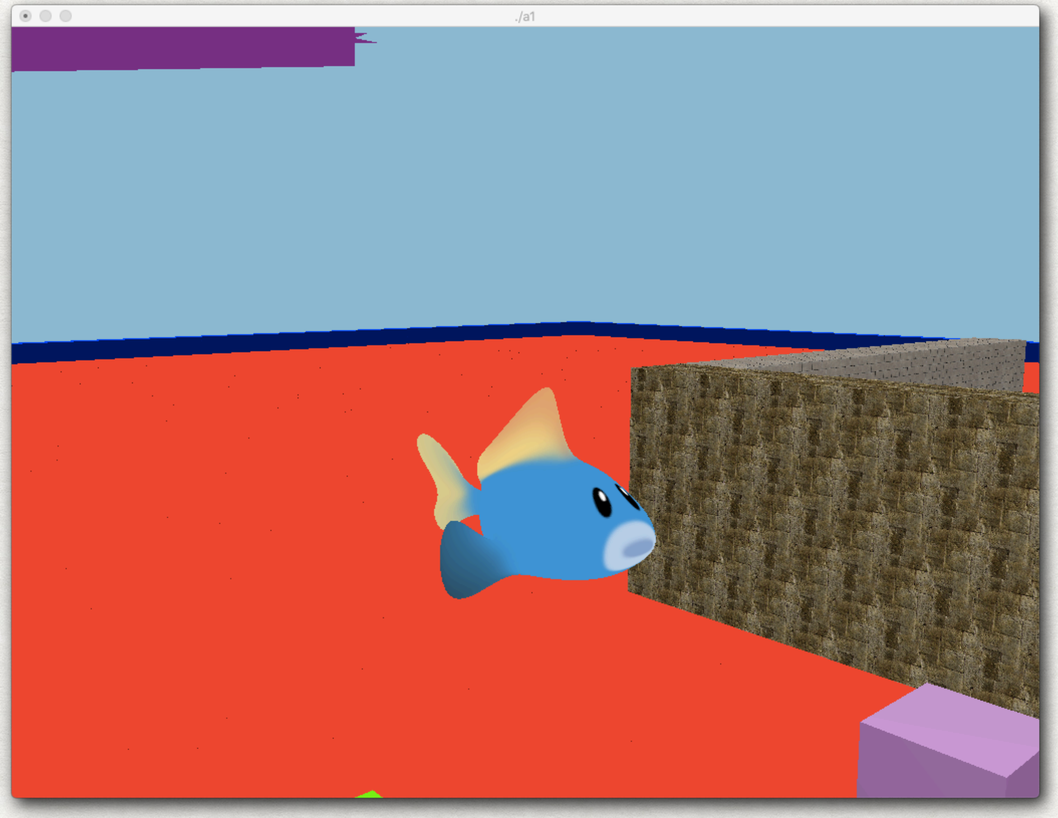
If an object cannot be seen by the user then the system should be told to not draw that object.

The game engine does this automatically for cubes. If you want to see the effect on the game when all cubes are drawn then run the game with the -drawall and -fps command line flags and compare the frames per second with and without using the -drawall option.

Create mesh objects in each room (the same ones used for animated mesh movement above) and test if they are visible to the user. If the object is visible then tell the game engine to draw the object. If it is not visible then hide the object.

VISIBILITY TESTING

The game currently comes with four mesh objects. The fish, cow, bat, and cactus. These can be loaded, positioned (translated), scaled, and rotated in the game world. There are also functions to enable and disable their drawing.



VISIBILITY TESTING

A mesh object will be put in each room of the maze.

As the player moves through the maze the system should turn on the visibility for an object when the player can see the object.

The system should print a message to the terminal indicating the object is visible:

```
Cow mesh #5 is visible.
```

When the player moves and the mesh is no longer visible then another message should be displayed.

```
Cow mesh #5 is not visible.
```

This message should only be printed after the mesh has been identified as visible.

There should be a visible state maintained and when the mesh is no longer visible the state should change. The system should not constantly print out messages about all of the meshes that cannot be seen.

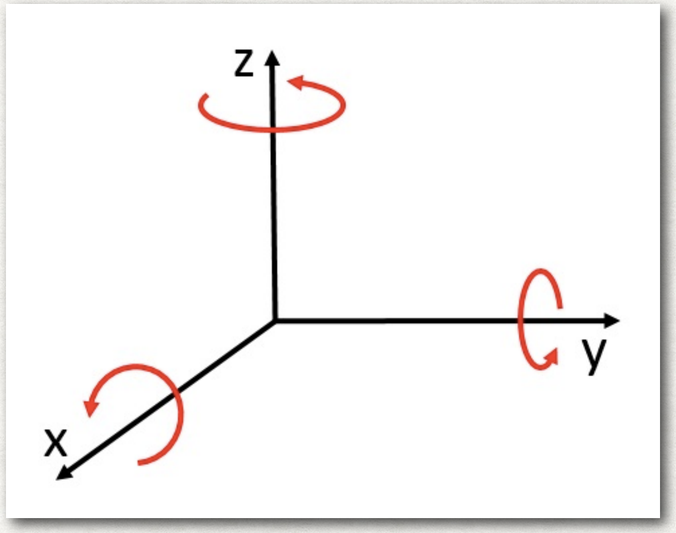
VISIBILITY TESTING

You can use the drawMesh() and hideMesh() functions to control if a mesh is drawn or not. Don't use unsetMeshID() as this will free remove that mesh from the list of active objects.

The game should end up with a mesh object in each room but only the ones which are near the viewer and in front of them will be drawn.

To determine if the user is facing towards an object you will need to calculate what is in front of the user. The system Euler Angles to indicate orientation of the viewer. These are a set of rotations around each of the three axis.

zRot



yRot

xRot

VISIBILITY TESTING

Given these three angles of rotation you can calculate the forward direction for the viewpoint using:

x = sin(yRot)
y = sin(xRot)
z = cos(yRot)

where (x, y, z) will be the new position. It is as a vector in the direction of the viewpoint which can be used for visibility testing or to calculate the next movement position.

Reverse motion can be calculated using negative values:

x = -sin(yRot)
y = -sin(xRot)
z = -cos(yRot)

VISIBILITY TESTING

Moving left and right (strafing) can be calculated similarly.

Left strafe:

x = cos(yRot)
z = sin(yRot)

Right strafe:

x = -cos(yRot)
z = -sin(yRot)

VISIBILITY TESTING

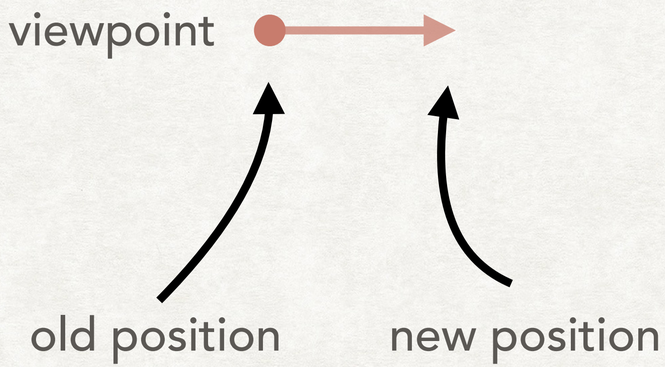
The game engine for the course uses this to determine motion based on key presses. The mouse movements are translated to Euler rotations and the next position of the viewpoint is calculated. You can calculate the vector showing the direction the user is facing by using the old and new position position values for (x,y,z).

```
case 'w':          // forward motion
         oldvpx = vpx;
         oldvpy = vpy;
         oldvpz = vpz;
         rotx = (mvx / 180.0 * 3.141592);    // convert to radians
         roty = (mvy / 180.0 * 3.141592);
         vpx -= sin(roty) * 0.3;
                // turn off y motion so you can't fly
         if (flycontrol == 1)
            vpy += sin(rotx) * 0.3;
         vpz += cos(roty) * 0.3;
         collisionResponse();
         glutPostRedisplay();
         break;
```

VISIBILITY TESTING

You can see that the old and new positions for the viewpoint in the game engine are a **viewing vector** pointing in the direction which the player can see.

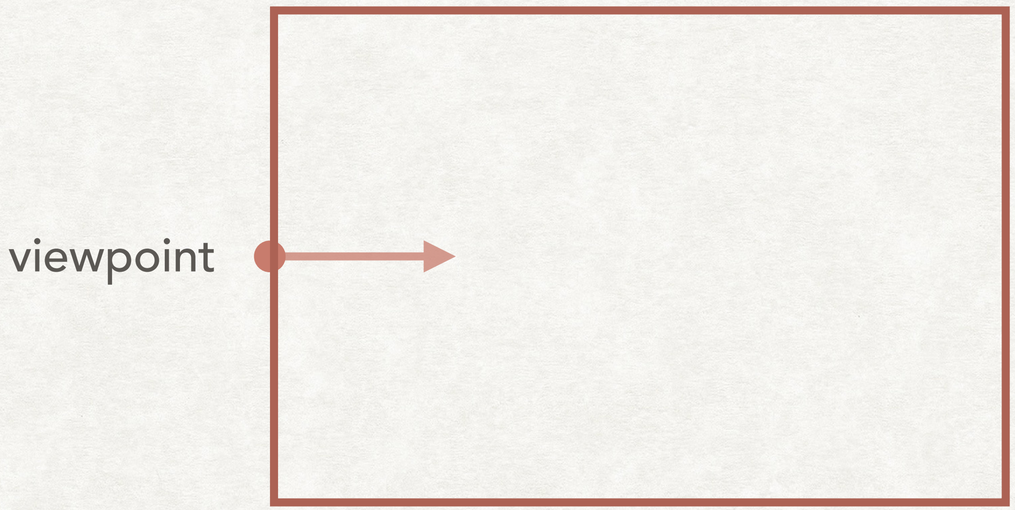
Use the direction vector to calculate the space in from of the user where mesh objects will be drawn. Objects outside of that space should not be drawn.



VISIBILITY TESTING

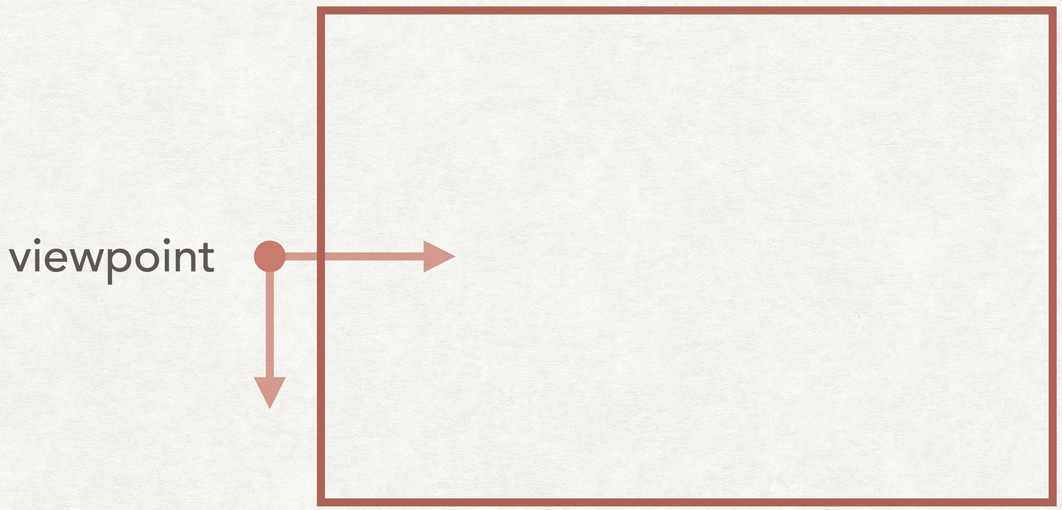
Determine the space in front of the viewpoint where the player can see.

You don’t need to consider space behind the viewer.



VISIBILITY TESTING

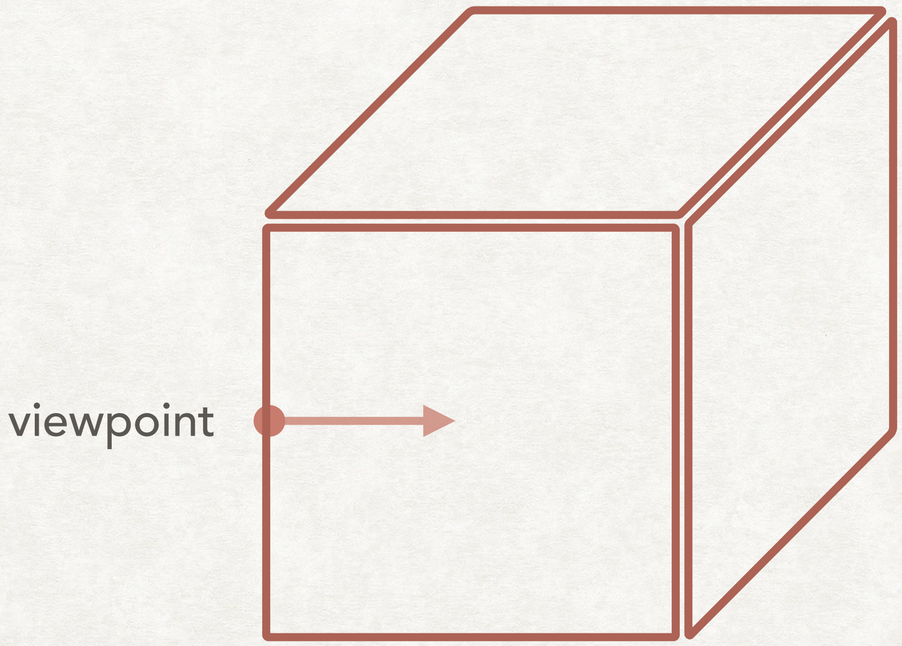
You can determine the orientation of the viewing volume by using the forward vector and a vector pointing to the side. With these you can determine the width and depth of the volume by multiplying them by the size of the cube. You can set the height to a fixed value because the cube will always be parallel to the ground (it only rotates around the y axis).



VISIBILITY TESTING

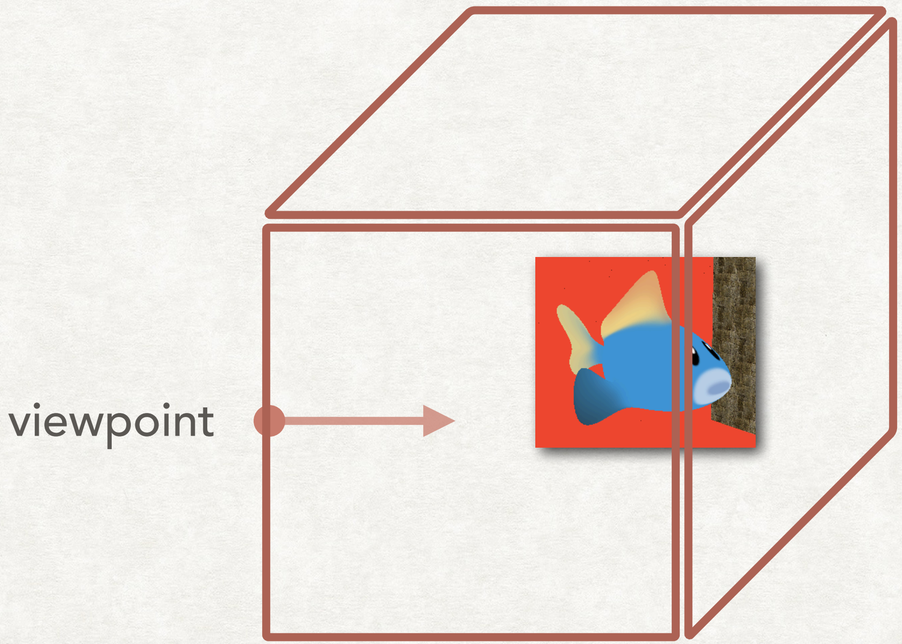
Determine the space in front of the viewpoint where the player can see. Pick dimensions for this space so the user cannot see when the mesh object is made visible or hidden. The transition should not be obvious.

This will be a three dimensional space. You will need to calculate a width, depth, and height.



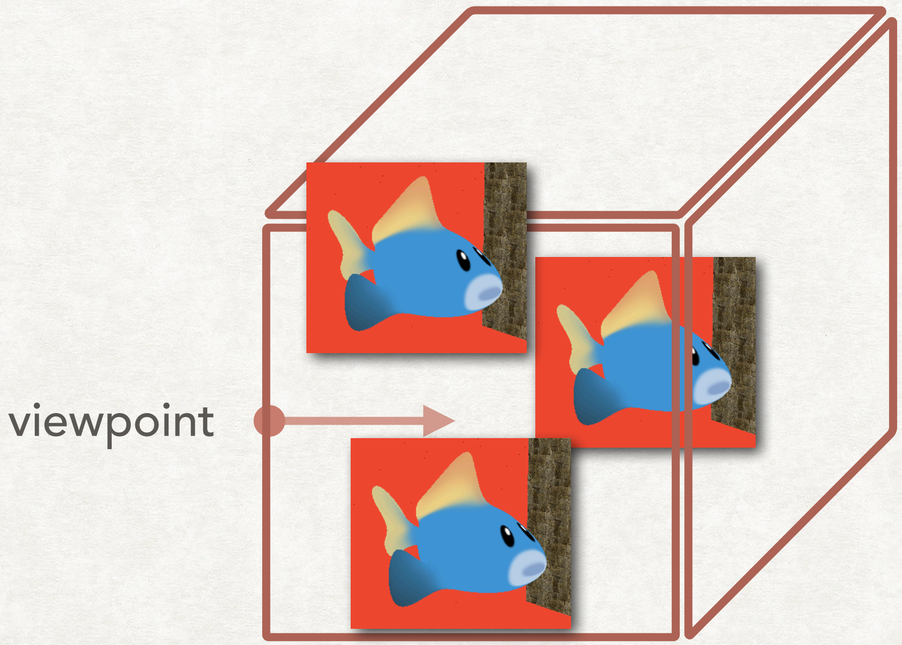
VISIBILITY TESTING

If the mesh object appears in the cube then it should be drawn.



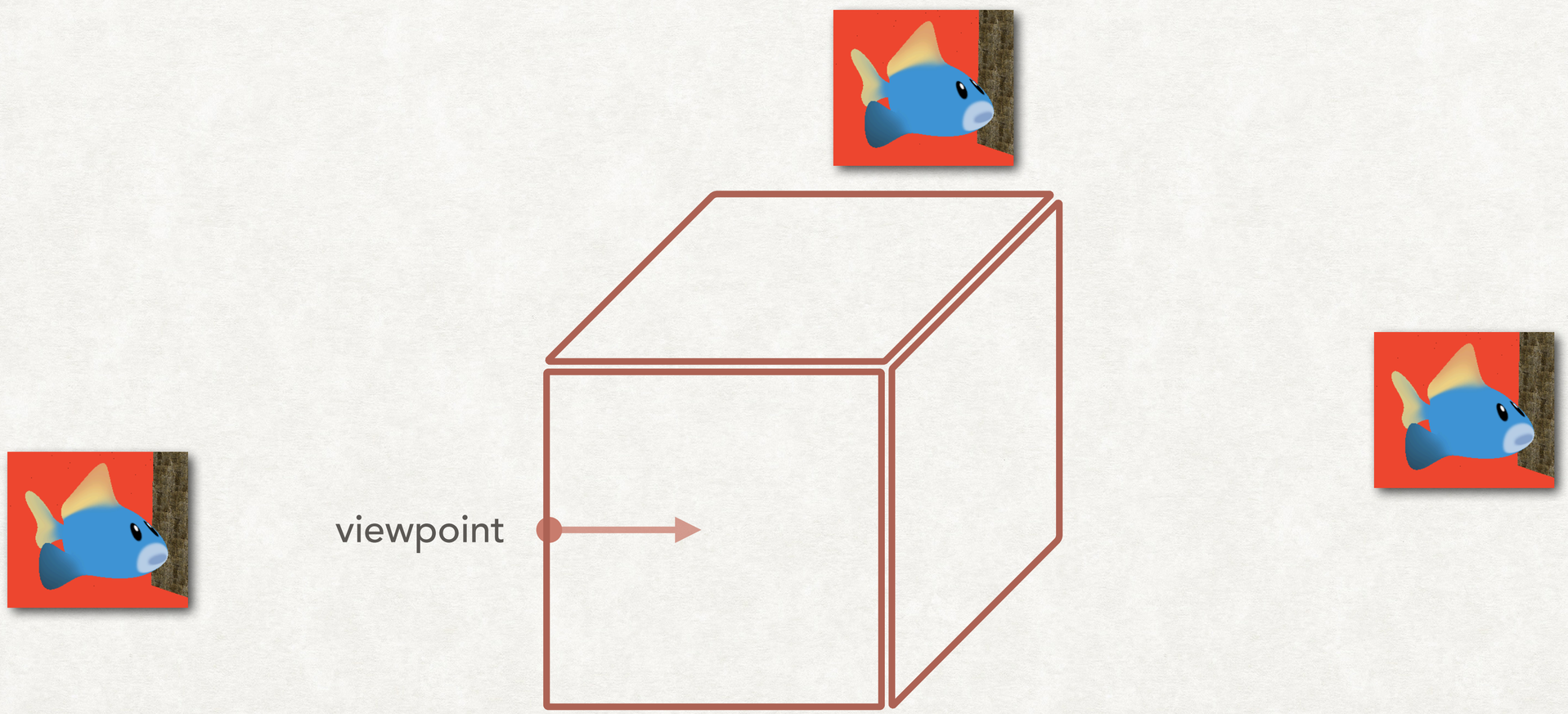
VISIBILITY TESTING

If the mesh object appears in the cube then it should be drawn. Multiple objects may appear in the viewing volume. In this case, all mesh objects should be drawn.



VISIBILITY TESTING

If the mesh object appears outside the cube then it should not be drawn.



VISIBILITY TESTING

If you make the viewing volume a cube then it is easy to determine if a mesh object appears inside the cube. Compare the mesh object's coordinates to the minimum and maximum values of the cube (these will be two opposite corners of the cube). If the mesh object's coordinates are all inside the cube then the mesh is visible.