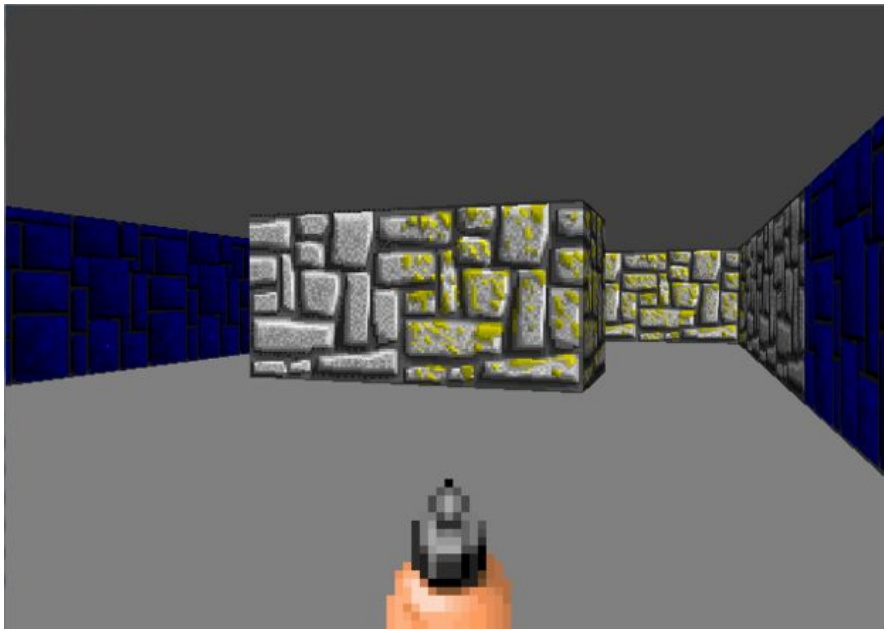

# Making a Basic 3D Engine in Java

by sheeptheelectric (/member/sheeptheelectric/)





(http://cdn.instructables.com/FJI/P8V1/I9GW6UE8/FJIP8V1I9GW6UE8.LARGE.jpg)

Having a game take place in a 3D environment greatly enhances the immersion, but actually implementing a full 3D engine can be very complex. Fortunately, there are some tricks that can be used to achieve the 3D effect in a relatively easy way. One of these tricks is called raycasting. Raycasting works by sending out a ray from the camera for each vertical bar on the screen and figuring out where that ray collides with a solid object. Raycasting is also very fast, and some of the first 3D games, like Wolfenstein 3D, used it. The engine described here is a very basic raycasting engine where all of the walls will be the same size and shape.

Making a raycasting engine is not too difficult, but it definitely requires some prior experience with programming. In addition to some prior programming experience I also recommend an IDE like Eclipse (https://eclipse.org/) or



## About This Instructable

**1,760** views

**56** favorites

License: (cc) BY-NC-SA

(/member/sheeptheelectric/) **sheeptheelectric (/member/sheeptheelectric/)**

Follow 1

**Tags:**

3D (/tag/type-id/category-technology/keyword-3d/)

Java (/tag/type-id/category-technology/keyword-java/)

Engine (/tag/type-id/category-technology/keyword-engine/)

Games (/tag/type-id/category-technology/keyword-games/)

Netbeans (https://netbeans.org/)

Programming (/tag/type-id/category-technology/keyword-programming

Code (/tag/type-id/category-technology/keyword-code/)

## Step 1: The Main Class

The first thing that needs to be made is a main class. The main class will handle displaying images to the user, calling on other classes to recalculate what should be displayed to the player, and updating the position of the camera.

For this class the imports will be:

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.image.BufferStrategy;
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferInt;
import java.util.ArrayList;
import javax.swing.JFrame;
```

The class and its variables will look like this:

```
public class Game extends JFrame implements Runnable{
        private static final long serialVersionUID = 1L;
        public int mapWidth = 15;
        public int mapHeight = 15;
        private Thread thread;
        private boolean running;
        private BufferedImage image;
        public int[] pixels;
        public static int[][] map =
                {
                        {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2},
                        {1,0,0,0,0,0,0,0,2,0,0,0,0,0,2},
                        {1,0,3,3,3,3,3,0,0,0,0,0,0,0,2},
                        {1,0,3,0,0,0,3,0,2,0,0,0,0,0,2},
                        {1,0,3,0,0,0,3,0,2,2,2,0,2,2,2},
                        {1,0,3,0,0,0,3,0,2,0,0,0,0,0,2},
                        {1,0,3,3,0,3,3,0,2,0,0,0,0,0,2},
                        {1,0,0,0,0,0,0,0,2,0,0,0,0,0,2},
                        {1,1,1,1,1,1,1,1,4,4,4,0,4,4,4},
                        {1,0,0,0,0,0,0,1,4,0,0,0,0,0,4},
                        {1,0,0,0,0,0,0,1,4,0,0,0,0,0,4},
                        {1,0,0,2,0,0,1,4,0,3,3,3,3,0,4},
                        {1,0,0,0,0,0,1,4,0,3,3,3,3,0,4},
                        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
                        {1,1,1,1,1,1,1,1,4,4,4,4,4,4,4}
```

Note that the map can be reconfigured to whatever you want, what I have here is merely a sample. The numbers on the map represent what type of wall will be at that position. A 0 represents empty space while any other number represents a solid wall and the texture that goes with it. The BufferedImage is what is displayed to the user, and pixels is an array of all the pixels in the image. The other variables won't really appear again, they are just used to get the graphics and program working properly.

The constructor will look like this for now:


### Related


**Using Blender To Create Java3D Models (/id/Using-Blender-To-Create-Java3D-Models/)**


**Learning Java: Your First Program! (/id/Learning-Java-Your-First-Program/)**
by CarterD (/member/CarterD/)


**Java Programming Part2(Text and running) (/id/Java-Programming-Part2Text-and-running/)**


**The basics of Programming (/id/The-basics-of-Programming/)**
by electronics for everyone


**How to Write First Program in JAVA (/id/How-to-Write-First-Program-in-JAVA/)**
by Kevin O Brien (/member

See More (/tag/type-id/?q=)

```
public Game() {
                thread = new Thread(this);
                image = new BufferedImage(640, 480, BufferedImage.TYPE_IN
T_RGB);
                pixels = ((DataBufferInt)image.getRaster().getDataBuffer(
)).getData();
                setSize(640, 480);
                setResizable(false);
                setTitle("3D Engine");
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                setBackground(Color.black);
                setLocationRelativeTo(null);
                setVisible(true);
                start();
        }
```

Most of this is just initialization of the class variables and the frame. The code after "pixels =" connects pixels and image so that any time the data values in pixels are changed the corresponding changes appear on the image when it is displayed to the user.

The start and stop methods are simple and used to make sure the program safely starts and ends.

```
private synchronized void start() {
        running = true;
        thread.start();
}
public synchronized void stop() {
        running = false;
        try {
                thread.join();
        } catch(InterruptedException e) {
                e.printStackTrace();
        }
}
```

The final two methods that need to be in the Game class are the render and run methods. The render method will look like this:

```
public void render() {
        BufferStrategy bs = getBufferStrategy();
        if(bs == null) {
                createBufferStrategy(3);
                return;
        }
        Graphics g = bs.getDrawGraphics();
        g.drawImage(image, 0, 0, image.getWidth(), image.getHeight(), nul
l);
        bs.show();
}
```

A buffer strategy is used when rendering so that screen updates are smoother. Overall, using a buffer strategy just helps the game look better when running. To actually draw the image to the screen a graphics object is obtained from the buffer strategy and used to draw our image.

The run method is very important because it handles how often different parts of the program are updated. To do this it uses some code to keep track of when 1/60th of a second has passed, and when it has the screen and camera are updated. This enhances how smoothly the program runs. The run method looks like this:

```
public void run(){
        long lastTime = System.nanoTime();
        final double ns = 1000000000.0 / 60.0;//60 times per second
        double delta = 0;
        requestFocus();
        while(running) {
                long now = System.nanoTime();
                delta = delta + ((now-lastTime) / ns);
                lastTime = now;
                while (delta >= 1)//Make sure update is only happening 60
 times a second
                {
                        //handles all of the logic restricted time

                        delta--;
                }
                render();//displays to the screen unrestricted time
        }
}
```

Once all of these methods, constructors, and variables are in then the only thing left to do in the Game class at the moment is to add a main method. The main method is very easy all you have to do is:

```
public static void main(String [] args) {
        Game game = new Game();
}
```

And now the main class is done for the moment! If you run the program now a black screen should pop up.

## Step 2: The Texture Class

(http://cdn.instructables.com/F31/4B9H/I9OSUEND/F314B9HI9OSUEND.LARGE.jpg)

(http://cdn.instructables.com/FIR/XLR9/I9OSUENE/FIRXLR9I9OSUENE.LARGE.jpg)

Before jumping into the calculations for finding how the screen should look I'm going to take a detour and set up a Texture class. Textures will be applied to the various walls in the environment, and will come from images saved in the project folder. In the images I have included 4 textures I found online that I will use in this project. You can use whatever textures you want. To use these textures I recommend putting them in a folder within the project file. To do this go to the project folder (in eclipse this is located in the workspace folder). After you get to the project folder create a new folder titled "res" or something. Place the textures in this folder. You can place the textures somewhere else, this is just where I store my textures. Once this is done we can start writing the code to make the

textures usable.

The imports for the class are:

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
```

The class header and its variables will look like this:

(http://cdn.instructables.com/FOI/9UFU/I9OSUENT/FOI9UFUI9OSUENT.LARGE.jpg)

```
public class Texture {
        public int[] pixels;
        private String loc;
        public final int SIZE;
```

The array pixels is used to hold the data for all the pixels in the image of the texture. Loc is used to indicate to the computer where the image file for the texture can be found. SIZE is how big the texture is on one side (a 64x64 image would have size 64), and all textures will be perfectly square.

The constructor will initialize the loc and SIZE variables and call the a method to load the image data into pixels. It looks like this:

```
public Texture(String location, int size) {
        loc = location;
        SIZE = size;
        pixels = new int[SIZE * SIZE];
        load();
}
```

Now all that's left for the Texture class is to add a load method to get data from images and store them in an array of pixel data. This method will look like this:

```
private void load() {
        try {
                BufferedImage image = ImageIO.read(new File(loc));
                int w = image.getWidth();
                int h = image.getHeight();
                image.getRGB(0, 0, w, h, pixels, 0, w);
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

The load method works by reading the data from the file that loc points to and writing this data to a buffered image. The data for every pixel is then taken from the buffered image and stored in pixels.

At this point the Texture class is done, so I'm going to go ahead and define a few textures that will be used in the final program. To do this put this

```
public static Texture wood = new Texture("res/wood.png", 64);
public static Texture brick = new Texture("res/redbrick.png", 64);
public static Texture bluestone = new Texture("res/bluestone.png", 64);
public static Texture stone = new Texture("res/greystone.png", 64);
```

between the "public class Texture" line and "public int[] pixels".

To make these textures accessible to the rest of the program let's go ahead and give them to the Game class. To do this we will need an ArrayList to hold all of the textures, and we will need to add the textures to this ArrayList. To create the ArrayList put the following line of code with the variables near the top of the class:

```
public ArrayList<Texture> textures;
```

This ArrayList will have to be initialized in the constructor, and the textures

should also be added to it in the constructor. In the constructor add the following bit of code:

```
textures = new ArrayList<Texture>();
textures.add(Texture.wood);
textures.add(Texture.brick);
textures.add(Texture.bluestone);
textures.add(Texture.stone);
```

And now textures are good to go!

## Step 3: The Camera Class

Now let's take another detour and set up the Camera class. The Camera class keeps track of where the player is located in the 2D map, and also takes care of updating the player's position. To do this the class will implement KeyListener, so it will need to import KeyEvent and KeyListener.

```
import java.awt.event.KeyEvent;<br>import java.awt.event.KeyListener;
```

Many variables are needed to keep track of the camera's position and what it can see. Because of this the first chunk of the class looks like this:

```
public class Camera implements KeyListener {
public double xPos, yPos, xDir, yDir, xPlane, yPlane;
        public boolean left, right, forward, back;
        public final double MOVE_SPEED = .08;
        public final double ROTATION_SPEED = .045;
```

xPos and yPos are the location of the player on the 2D map that was created in the Game class. xDir and yDir are the x and y components of a vector that points in the direction the player is facing. xPlane and yPlane are also the x and y components of a vector. The vector defined by xPlane and yPlane is always perpendicular to the direction vector, and it points to the farthest edge of the camera's field of view on one side. The farthest edge on the other side is just the negative plane vector. The combination of the direction vector and the plane vector defines what is in the camera's field of view. The booleans are used to keep track of what keys are being pressed by the user so that the user can move the camera. MOVE_SPEED and ROTATION_SPEED dictate how quickly the camera moves and turns while the user is pressing the corresponding key.

Next is the constructor. The constructor takes in values that tell the class where the camera is located and what it can see and assigns them to the corresponding variable (xPos, yPos...).

```
public Camera(double x, double y, double xd, double yd, double xp, double
 yp)
{
        xPos = x;
        yPos = y;
        xDir = xd;
        yDir = yd;
        xPlane = xp;
        yPlane = yp;
}
```

A camera object will be needed in the final program, so let's go ahead and add one. In the Game class with all of the other variable declarations add in

```
public Camera camera;
```

and in the constructor add in

```
camera = new Camera(4, 6, 1, 0, 0, -.66);
addKeyListener(camera);
```

This camera will work with the map I am using, if you are using a different map or if you want to start in a different location adjust the values of xPos and yPos (4 and 6 in my example). Using .66 gives what I feel is a good field of vision, but you can adjust the value to get a different FOV.

Now that the Camera class has a constructor we can start adding methods to track the inputs of the user and to update the position/orientation of the camera. Because the Camera class implements KeyboardListener it must have all the methods from it implemented. Eclipse should automatically prompt you to add these methods. You can leave the keyTyped method blank, but the other two methods will be used. keyPressed will set the booleans to true when their corresponding keys are pressed, and keyReleased will change them back to false when the keys are released. The methods look like this:

```
public void keyPressed(KeyEvent key) {
        if((key.getKeyCode() == KeyEvent.VK_LEFT))
                left = true;
        if((key.getKeyCode() == KeyEvent.VK_RIGHT))
                right = true;
        if((key.getKeyCode() == KeyEvent.VK_UP))
                forward = true;
        if((key.getKeyCode() == KeyEvent.VK_DOWN))
                back = true;
}
```

and

```
public void keyReleased(KeyEvent key) {
        if((key.getKeyCode() == KeyEvent.VK_LEFT))
                left = false;
        if((key.getKeyCode() == KeyEvent.VK_RIGHT))
                right = false;
        if((key.getKeyCode() == KeyEvent.VK_UP))
                forward = false;
        if((key.getKeyCode() == KeyEvent.VK_DOWN))
                back = false;
}
```

Now that the Camera class is keeping track of which keys are pressed we can start updating the player's position. To do this we will use an update method that is called in the run method of the Game class. While we are at it we'll go ahead and add collision detection to the update method by passing the map to it when it is called in the Game class. The update method looks like this:

```
public void update(int[][] map) {
        if(forward) {
                if(map[(int)(xPos + xDir * MOVE_SPEED)][(int)yPos] == 0)
{
                        xPos+=xDir*MOVE_SPEED;
                }
                if(map[(int)xPos][(int)(yPos + yDir * MOVE_SPEED)] ==0)
                        yPos+=yDir*MOVE_SPEED;
        }
        if(back) {
                if(map[(int)(xPos - xDir * MOVE_SPEED)][(int)yPos] == 0)
                        xPos-=xDir*MOVE_SPEED;
                if(map[(int)xPos][(int)(yPos - yDir * MOVE_SPEED)]==0)
                        yPos-=yDir*MOVE_SPEED;
        }
        if(right) {
                double oldxDir=xDir;
                xDir=xDir*Math.cos(-ROTATION_SPEED) - yDir*Math.sin(-ROTA
TION_SPEED);
                yDir=oldxDir*Math.sin(-ROTATION_SPEED) + yDir*Math.cos(-R
OTATION_SPEED);
                double oldxPlane = xPlane;
                xPlane=xPlane*Math.cos(-ROTATION_SPEED) - yPlane*Math.sin
(-ROTATION_SPEED);
                yPlane=oldxPlane*Math.sin(-ROTATION_SPEED) + yPlane*Math.
cos(-ROTATION_SPEED);
```

The parts of the method that control forward and backwards movement work by adding xDir and yDir to xPos and yPos, respectively. Before this movement happens the program checks if the movement will put the camera inside a wall, and doesn't go through with the movement if it will. For rotation both the direction vector and the plane vector are multiplied by the rotation matrix, which is:

```
[ cos(ROTATION_SPEED) -sin(ROTATION_SPEED) ]
[ sin(ROTATION_SPEED)  cos(ROTATION_SPEED) ]
```

to get their new values. With the update method completed we can now call it from the Game class. In the Game class' run method add the following line of code where it is shown here

```
Add this:
camera.update(map);
in here:
while(running) {
        long now = System.nanoTime();
        delta = delta + ((now-lastTime) / ns);
        lastTime = now;
        while (delta >= 1)//Make sure update is only happening 60 times a
 second
        {
                //handles all of the logic restricted time
                camera.update(map);
                delta--;
        }
        render();//displays to the screen unrestricted time
}
```

Now that that's done we can finally move onto the final class and calculate the screen!

## Step 4: Calculating the Screen

The Screen class is where the majority of the calculations are done to get the program working. To work, the class needs the following imports:

```
import java.util.ArrayList;
import java.awt.Color;
```

The actual class begins like this:

```
public class Screen {
    public int[][] map;
    public int mapWidth, mapHeight, width, height;
    public ArrayList textures;
```

The map is the same map created in the game class. The screen uses this to figure out where walls are and how far away from the player they are. Width and height define the size of the screen, and should always be the same as the width and height of the frame created in the Game class. Textures is a list of all the textures so that the screen can access the pixels of the textures. After those variables are declared they have to be initialized in the constructor like so:

```
public Screen(int[][] m, ArrayList tex, int w, int h) {
    map = m;
    textures = tex;
    width = w;
    height = h;
}
```

Now its time to write the one method the class has: an update method. The update method recalculates how the screen should look to the user based on their position in the map. The method is called constantly, and returns the updated array of pixels to the Game class. The method begins by "clearing" the screen. It does this by setting all of the pixels on the top half to one color and all of the pixels on the bottom to another.

```
public int[] update(Camera camera, int[] pixels) {
    for(int n=0; n<pixels.length/2; n++) {
        if(pixels[n] != Color.DARK_GRAY.getRGB()) pixels[n] = Color.DARK_
GRAY.getRGB();
    }
    for(int i=pixels.length/2; i<pixels.length; i++) {
        if(pixels[i] != Color.gray.getRGB()) pixels[i] = Color.gray.getRG
B();
    }
```

Having the top and bottom of the screen be two different colors also makes it seem like there is a floor and a ceiling. After the pixel array is cleared then it is time to move onto the main calculations. The program loops through every vertical bar on the screen and casts a ray to figure out what wall should be on the screen at that vertical bar. The beginning of the loop looks like this:

```
for(int x = 0; x < width; x++){
    double cameraX = 2 * x / (double)(width) -1;
    double rayDirX = camera.xDir + camera.xPlane * cameraX;
    double rayDirY = camera.yDir + camera.yPlane * cameraX;
    //Map position
    int mapX = (int)camera.xPos;
    int mapY = (int)camera.yPos;
    //length of ray from current position to next x or y-side
    double sideDistX;
    double sideDistY;
    //Length of ray from one side to next in map
    double deltaDistX = Math.sqrt(1 + (rayDirY*rayDirY) / (rayDirX*rayDir
X));
    double deltaDistY = Math.sqrt(1 + (rayDirX*rayDirX) / (rayDirY*rayDir
Y));
    double perpWallDist;
    //Direction to go in x and y
    int stepX, stepY;
    boolean hit = false;//was a wall hit
    int side=0;//was the wall vertical or horizontal
```

All that happens here is some variables that will be used by the rest of the loop are calculated. CameraX is the x-coordinate of the current vertical stripe on the camera plane, and the rayDir variables make a vector for the ray. All of the variables ending in DistX or DistY are calculated so that the program only checks for collisions at the places where collisions could possibly occur. perpWallDist is the distance from the player to the first wall the ray collides with. This will be calculated later. After that is done we need to figure out a few of the other variables based on the one we already calculated.

```
//Figure out the step direction and initial distance to a side
if (rayDirX < 0)
{
    stepX = -1;
    sideDistX = (camera.xPos - mapX) * deltaDistX;
}
else
{
    stepX = 1;
    sideDistX = (mapX + 1.0 - camera.xPos) * deltaDistX;
}
if (rayDirY < 0)
{
    stepY = -1;
    sideDistY = (camera.yPos - mapY) * deltaDistY;
}
else
{
    stepY = 1;
    sideDistY = (mapY + 1.0 - camera.yPos) * deltaDistY;
}
```

Once that is done it is time to figure out where the ray collides with a wall. To do this the program goes through a loop where it checks if the ray has come into contact with a wall, and if not moves to the next possible collision point before checking again.

```
//a loop to find where the ray hits a wall
while(!hit) {
    //Jump to next square
    if (sideDistX < sideDistY)
    {
        sideDistX += deltaDistX;
        mapX += stepX;
        side = 0;
    }
    else
    {
        sideDistY += deltaDistY;
        mapY += stepY;
        side = 1;
    }
    //Check if ray has hit a wall
    if(map[mapX][mapY] > 0) hit = true;
}
```

Now that we know where the ray hits a wall we can start figuring out how the wall should look in the vertical stripe we are currently on. To do this we first calculate the distance to the wall, and then use that distance to figure out how tall the wall should appear in the vertical strip. We then translate that height to a start and finish in terms of the pixels on the screen. The code looks like this:

```
//Calculate distance to the point of impact
if(side==0)
    perpWallDist = Math.abs((mapX - camera.xPos + (1 - stepX) / 2) / rayD
irX);
else
    perpWallDist = Math.abs((mapY - camera.yPos + (1 - stepY) / 2) / rayD
irY);
//Now calculate the height of the wall based on the distance from the cam
era
int lineHeight;
if(perpWallDist > 0) lineHeight = Math.abs((int)(height / perpWallDist));
else lineHeight = height;
//calculate lowest and highest pixel to fill in current stripe
int drawStart = -lineHeight/2+ height/2;
if(drawStart < 0)
    drawStart = 0;
int drawEnd = lineHeight/2 + height/2;
if(drawEnd >= height)
    drawEnd = height - 1;
```

After that is calculated it is time to begin figuring out what pixels from the texture of the wall will actually appear to the user. For this we first must figure out what texture is associated with the wall we just hit and then figure out the x-coordinate on the texture of the pixels that will appear to the user.

```
//add a texture
int texNum = map[mapX][mapY] - 1;
double wallX;//Exact position of where wall was hit
if(side==1) {//If its a y-axis wall
    wallX = (camera.xPos + ((mapY - camera.yPos + (1 - stepY) / 2) / rayD
irY) * rayDirX);
} else {//X-axis wall
    wallX = (camera.yPos + ((mapX - camera.xPos + (1 - stepX) / 2) / rayD
irX) * rayDirY);
}
wallX-=Math.floor(wallX);
//x coordinate on the texture
int texX = (int)(wallX * (textures.get(texNum).SIZE));
if(side == 0 && rayDirX > 0) texX = textures.get(texNum).SIZE - texX - 1;
if(side == 1 && rayDirY < 0) texX = textures.get(texNum).SIZE - texX - 1;
```

The x-coordinate is calculated by taking the exact position of where the wall was

hit on the 2D map and subtracting the integer value, leaving only the decimal. This decimal (wallX) is then multiplied by the size of the texture of the wall to get the exact x-coordinate on the wall of the pixels we wish to draw. Once we know that the only thing left to do is calculate the y-coordinates of the pixels on the texture and draw them on the screen. To do this we loop through all of the pixels on the screen in the vertical strip we are doing calculations for and calculate the the exact y-coordinate of the pixel on the texture. Using this the program then writes the data from the pixel on the texture into the array of pixels on the screen. The program also makes horizontal walls darker than vertical walls here to give a basic lighting effect.

```
//calculate y coordinate on texture
for(int y=drawStart; y<drawEnd; y++) {
    int texY = (((y*2 - height + lineHeight) << 6) / lineHeight) / 2;
    int color;
    if(side==0) color = textures.get(texNum).pixels[texX + (texY * textur
es.get(texNum).SIZE)];
    else color = (textures.get(texNum).pixels[texX + (texY * textures.get
(texNum).SIZE)]>>1) & 8355711;//Make y sides darker
    pixels[x + y*(width)] = color;
}
```

After that, all that is left in the Screen class is to return the pixel array

```
return pixels;
```

And the class is done. Now all we have to do is add a few lines of code in the Game class to get the screen working. With the variables at the top add this:

```
public Screen screen;
```

And in the constructor add this somewhere after textures has been initialized.

```
screen = new Screen(map, mapWidth, mapHeight, textures, 640, 480);
```

And finally, in the run method add

```
screen.update(camera, pixels);
```

right before camera.update(map). And the program is done!

## Step 5: The Final Code

Here's a complete copy of the code for each class.

**Game.txt** (/files/orig/FVH/UBXH/I9NE11F1/FVHUBXHI9NE11F1.txt)2 KB

**Texture.txt** (/files/orig/FOX/L5C7/I9NE11F2/FOXL5C7I9NE11F2.txt)891 bytes

**Camera.txt** (/files/orig/F6V/VL06/I9NE11F5/F6VVL06I9NE11F5.txt)2 KB

**Screen.txt** (/files/orig/FQR/07XM/I9NE11F6/FQR07XMI9NE11F6.txt)4 KB

We have a **be nice** comment policy.
Please be positive and constructive.

I Made it! Add Images Make Comment

(/member /nancyjohns/)

**nancyjohns (/member/nancyjohns/)** 3 months ago Reply

This is great. Where did you get the textures for the cover picture? Did you make them? I like the way they look.



(/member /sheeptheelectric/)

**sheeptheelectric (/member/sheeptheelectric/)** (author) nancyjohns 3 months ago Reply

The textures in the cover photo are from Wolfenstein 3D. You can probably find them somewhere on the Internet.

(/member /nancyjohns/)

**nancyjohns (/member/nancyjohns/)** sheeptheelectric 3 months ago Reply

Thanks!



(/member /01001011/)

**01001011 (/member/01001011/)** 3 months ago Reply

Wow, this is amazing. I want do this with C#, if a do something i will make a instructable :)
Great job!

(/member /amberrayh/)

**amberrayh (/member/amberrayh/)** 4 months ago Reply

Great job on your first Instructable! I hope we see more from you in the future!

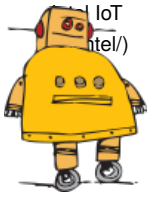
(/member /PaleHorseRider/)

**PaleHorseRider (/member/PaleHorseRider/)** 4 months ago Reply

And so you choose the copyrighted image of Wolfenstien...

## FEATURED CHANNELS

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| l IoT ntel/) | Life Hacks (/tag/type-id/category-home/channel-life-hacks/) | Beauty (/tag/type-id/category-home/channel-beauty/) | Woodworking (/tag/type-id/category-workshop/channel-woodworking/) | Minecraft (/tag/type-id/category-play/channel-minecraft/) | Breakfast (/tag/type-id/category-food/channel-breakfast/) | Laser Cut (/tag/type-id/category-workshop/channel-laser-cutting/) | Organizing (/tag/type-id/category-home/channel-organizing/) | Arduino (/tag/type-id/category-technology/channel-arduino/) |



**Newsletter**

Join 2 million + to receive instant inspiration in your inbox.

enter email I'm in!

### Mobile

Download our apps!

Android » (https://play.google.com/store/apps/details?id=com.adsk.instructables)
iOS » (https://itunes.apple.com/app/instructables/id586765571)
Windows » (http://apps.microsoft.com/windows/en-us/app/7afc8194-c771-441a-9590-54250d6a8300)

### About Us

Who We Are (/about/)
Advertise (/advertise/)
Contact (/about/contact.jsp)
Jobs (/community/Positions-available-at-Instructables/)
Help (/id/how-to-write-a-great-instructable/)

### Find Us

Facebook (http://www.facebook.com/instructables)
Youtube (http://www.youtube.com/user/instructablestv)
Twitter (http://www.twitter.com/instructables)
Pinterest (http://www.pinterest.com/instructables)
Google+ (https://plus.google.com/+instructables)
Tumblr (http://instructables.tumblr.com)

### Resources

For Teachers (/teachers/)
Artists in Residence (/air)
Gift Pro Account (/account/give?sourcea=footer)
Forums (/community/)
Answers (/tag/type-question/?sort=RECENT)
Sitemap (/sitemap/)

---