

# XNA DEVELOPMENT: TUTORIAL 3

BY MATTHEW CHRISTIAN (MATT@INSIDEGAMER.ORG)

Code and Other Tutorials Found at <http://www.insidegamer.org/xnatutorials.aspx>

After analyzing the basics, we're finally ready to work on getting stuff into our application. We'll be focusing on the technical side of game programming in this series but remember, you should design a bit before starting any project you would like to go anywhere with. If you're simply following these tutorials I've done the design for you though have designed it in a way where we will create tech demos utilizing certain features instead of full fledged games. Going back to this specific tutorial, 2D is nice, but 3D is better and we will be starting with 3D. Most new game developers want to create 3D games though are discouraged with the difficulty of it. Worry not, XNA will help us transition straight into 3D (we'll see 2D later anyways). Lets focus on 3D, importing a mesh, and displaying the mesh to the screen.

**Note:** 3D is almost completely based on mathematical topics such as linear algebra (matrices, matrix operations, etc...) so hopefully you've had some kind of background in the subject. If not, I'll try my best to provide you with the basics to get through this tutorial. But, I suggest you begin to learn linear algebra, it'll make everything easier for you in the long run.

## BEFORE YOU START

Following the first two tutorials you should be able to use the same project if you'd like, otherwise make sure you start a new project and create it with the Windows Game (2.0) template. I will be using the same project from the previous tutorials (WindowsGame1) and will be extending this project to include a new class. You can change the names of the files if you'd like but for ease of reference I use default names in this tutorial such as Game1.cs and Program.cs.

## GAMEMODEL CLASS DESIGN

As noted above, we will be creating a new class which will have three very basic features:

1. Load a specified model
2. Place the model in our game world
3. Render the object

You should be able to notice at least 4 methods of this class (including the constructor and destructor) and possibly have some kind of an idea what you want to name them and what they will take as method parameters. In fact, we will only have 4 total methods in our GameModel class (one is overloaded) (Diagram 1).

In future tutorials we will modify this class and the methods and variables contained within. This is simply a very basic class we're writing to draw a model to the screen and should be suitable enough for our needs.

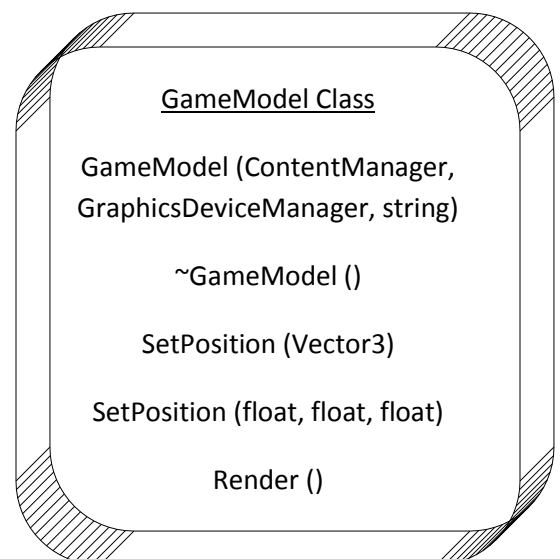


Diagram 1 – GameModel Class

## PROGRAMMING THE GAMEMODEL CLASS

Add a new class to your project by right clicking on the project name, scrolling to Add, and selecting from New Item... from the menu that appears. A box resembling the template selection when starting a new project will appear (Figure 1). Select Class from the Visual Studio Installed Templates area and change the name (mine will be named GameModel.cs).

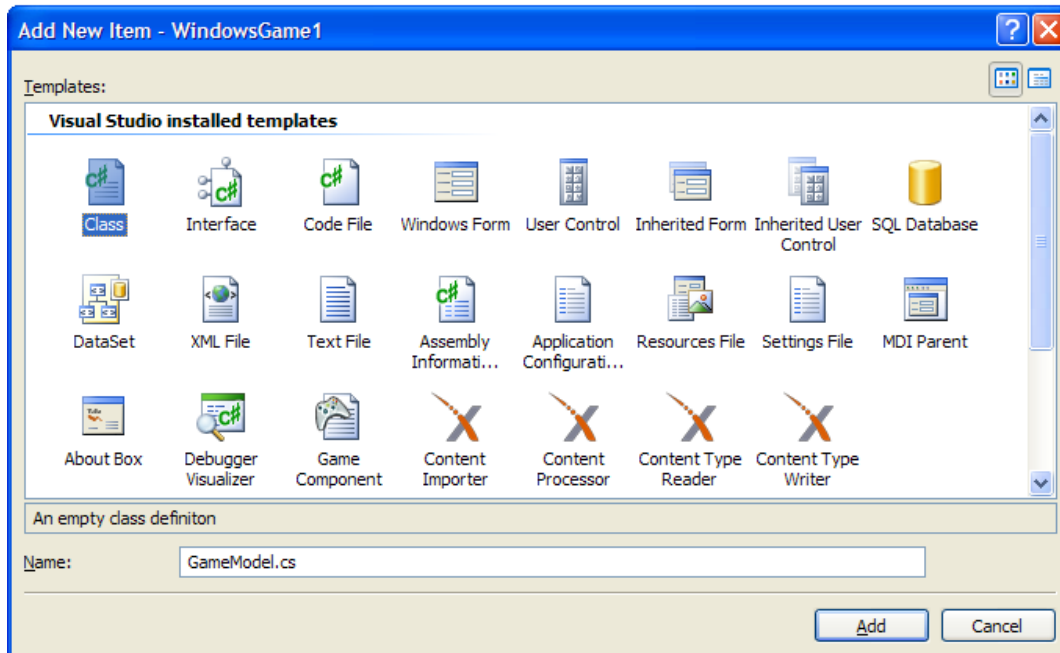


Figure 1 - Add New Item Dialogue

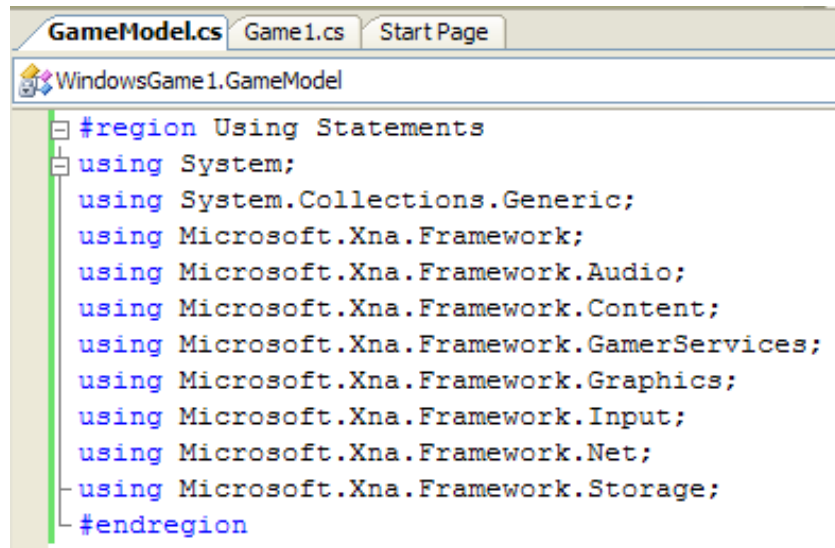
After selecting Add, the item should appear in your Solution Explorer and should open in the editor window as the file you are currently working with. Near the top of the file you should see a group of 'using' statements which include the system and XNA libraries. Your game file (default is Game1.cs) also has this block of code but has region tags placed around it which allow the programmer to 'collapse' the lines of code. To get this effect, place the following line of code above the line `using System;`

```
#region Using Statements
```

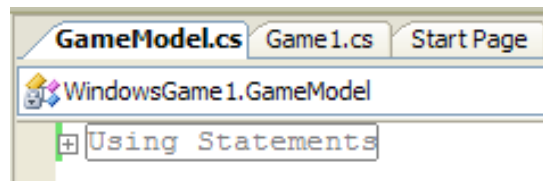
Here we're specifying what is called the region directive. It allows us to section off blocks of code and expand or collapse that block while editing our file in Visual Studio. The region directive needs to be terminated with another line (otherwise the computer won't ever find the end of the region and you will receive a compiler error). After the line `using Microsoft.Xna.Framework.Storage;`, add the following

```
#endregion
```

Now we've created a new region in our code and can expand it or collapse it as necessary (Figure 2).



```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
#endregion
```



```
GameModel.cs Game1.cs Start Page
WindowsGame1.GameModel
Using Statements
```

Figure 2 - Using Statements Region

Let's start working on the GameModel class. Create basic layouts for each of your functions by following the code below and placing it within your beginning and ending GameModel class tags (for a challenge, try creating the basic definitions based on the class diagram on Page 3 and check your answer against the below code).

```
public GameModel(ContentManager content, GraphicsDeviceManager graphics, string
    modelName)
{
    //Set managers and load our model
}

~GameModel() { }

public void SetPosition(Vector3 position)
{
    //Set Position using a Vector
}

public void SetPosition(float x, float y, float z)
{
    //Set Position using floats
}

public void Render()
{
    //Render our model using position data
}
```

If you don't understand this code, don't worry, let's go over it. Starting from the top we have our class constructor which we need to declare as public so we can create new objects of this class within our main file

(you'll hear lots more about objects later on). The constructor takes 3 arguments, a ContentManager, a GraphicsDeviceManager, and a string. Our GraphicsDeviceManager is created in our initial game file, Game1.cs, and used (in this context) to determine our screen's aspect ratio respectively. In XNA 2.0, the ContentManager is built directly into the game project and can be accessed in the Game1 file using the 'this' operator. The ContentManager is used to load in content through the XNA content pipeline (load the model being used). Finally, our method takes a string which will be asking for the model we want to load. In this method we want to initialize the rest of the class with values we'll need later. The constructor will set some global class variables with this data using the following code.

```
public GameModel(ContentManager content, GraphicsDeviceManager graphics, string
    modelName)
{
    //Load model and set graphics manager
    gModel = content.Load<Model>(modelName);

    //Calculate and set aspect ratio
    aspectRatio = graphics.GraphicsDevice.Viewport.Width /
        graphics.GraphicsDevice.Viewport.Height;

    //Default position
    positionVector = new Vector3(0.0f, 0.0f, 0.0f);
}
```

We have 3 new global variables, gModel, aspectRatio, and positionVector. All help the model define what it is and where it is but we will need them later in the Render() method which is why we set them globally in the class. Notice how a model is loaded using the ContentManager passed in, specifying the type of object being loaded, and specifying the name of the model (which we also passed in). Following the load is a quick calculation for our aspect ratio by dividing the screen width by the screen height and finishing the constructor by defaulting the position of the model to (0,0,0). Remember to create global class variables (typically above all your methods within the class) like the following (the last two matrices will be used later on),

```
//Class model storage
private Model gModel;

//Aspect Ratio
private float aspectRatio = 0.0f;

//Position of our model
private Vector3 positionVector;

//Projection and view matrix
private Matrix projection;
private Matrix view;
```

It may seem odd that you're sending your model name to this class only to set it to a Model (gModel) and you can certainly put this in your main file (Game1.cs) but creating this class will make the code much cleaner when you see the Render() method.

**Note:** I won't go into the destructor mainly because we don't do anything with it. It's simply added as a habit I've developed but helps round out the class a bit more. You don't need to include this method but if you'd like, place the following code within your class:

```
~GameModel() { }
```

## SETPosition()

Set position will be used to set the position of our model using our global positionVector variable. This method will be overloaded, meaning there will be two different methods of this class named SetPosition() but each will use different parameters which will tell the compiler which method to use. The easiest way to set the position vector would be to set it equal to another vector which is exactly what our first version of this method is.

```
public void SetPosition(Vector3 position)
{
    //Set Position using a Vector
    positionVector = position;
}
```

All that's needed since this version is receiving the same type is set them equal to each other. The problem arises when you don't have a Vector3 to pass to SetPosition(). When we have something like a camera we would probably have a vector but if we want to just specify a position it would be a hassle to create a vector, set it to a position, and then pass the vector to our GameModel class. Instead, we'll create an overloaded method of SetPosition() that will accept 3 floats and will set them to our X, Y, and Z positions of the positionVector variable. Here is the implementation of this method:

```
public void SetPosition(float x, float y, float z)
{
    //Set Position using floats
    positionVector.X = x;
    positionVector.Y = y;
    positionVector.Z = z;
}
```

XNA makes working with vectors and the separate values of them really easy to define and set. All that's needed is 3 floats and we can access the X, Y, and Z of the positionVector and set them to the floats that are passed in.

## RENDER()

The Render() method of our GameModel class is the largest and potentially most confusing method in this class. It also is the primary reason for creating a specific class for each model because instead of writing over 15 lines of code in our game's primary Draw() method we'll only need to call one. Here's the full code for our Render() method, we'll discuss it line by line after.

```
public void Render()
{
    Matrix[] transforms = new Matrix[gModel.Bones.Count];
    gModel.CopyAbsoluteBoneTransformsTo(transforms);

    projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
        aspectRatio, 1.0f, 10000.0f);
    view = Matrix.CreateLookAt(Vector3.Zero, positionVector, Vector3.Up);

    //Draw every mesh in the model
    foreach (ModelMesh mesh in gModel.Meshes)
    {
        //Set our mesh orientation, camera, and projection
        foreach (BasicEffect effect in mesh.Effects)
        {
            //Run default lighting
            effect.EnableDefaultLighting();
        }
    }
}
```

```

        //Model Matrices
        effect.View = view;
        effect.Projection = projection;
        effect.World = transforms[mesh.ParentBone.Index] *
            Matrix.CreateTranslation(positionVector);
    }

    //Draw the mesh
    mesh.Draw();
}
}

```

This method looks confusing but most of it can be broken down into simple pieces.

```

Matrix[] transforms = new Matrix[gModel.Bones.Count];
gModel.CopyAbsoluteBoneTransformsTo(transforms);

```

The method starts with creating a local matrix called transforms and we send it the amount of bones in our model. Following we copy bones from our model in to our transforms matrix. These lines are only necessary when a more complicated model is used that has its position, scale, and rotation defined by bones within the model. If no such bone structure exists the code simply doesn't use it.

```

projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
    aspectRatio, 1.0f, 10000.0f);
view = Matrix.CreateLookAt(Vector3.Zero, positionVector, Vector3.Up);

```

Next we set two of our class's global variables, the projection and view matrix variables. We need to orient our model and camera in the game world and these variables allow us to do that. The projection matrix creates a view frustum and accepts floats for a field of view (defined in radians, hence using MathHelper to convert 45.0 degrees to radians), the aspect ratio (remember when we set this in the class constructor?), the near plane distance, and the far plane distance. A frustum is what the camera sees and can best be described as a sectioned, 5-sided pyramid on its side (Figure 3).

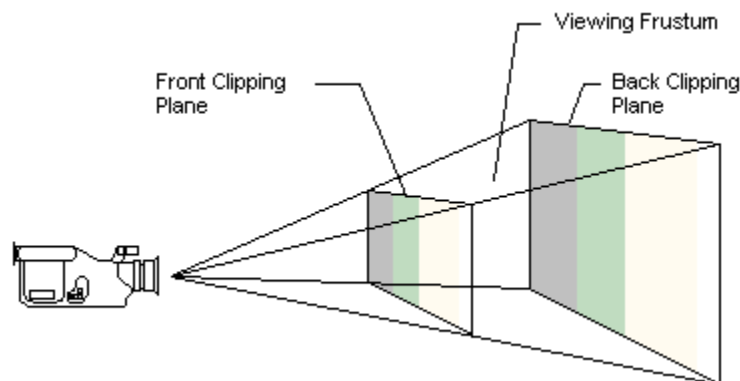


Figure 3 - Camera View Frustum

You can see where the front clipping plane and back clipping plane are located and everything in between is our view frustum. The game camera will render anything in the view frustum and will disregard anything outside of it. In fact, this leads to some drawing and loading problems that I won't cover here but will go into more in advanced camera and view tutorials.

Our Render() code has our front (or near) clipping plane 1.0f in front of the camera position and our back (or far) clipping plane at 10000.0f. Obviously if you want to see further into the level just extend the far plane. Second, we set our view matrix using the CreateLookAt() method of the Matrix class predefined in XNA. CreateLookAt() sets the way our camera is looking by requiring a camera position, a target (or where we're

looking at), and the world's up direction. I've set the camera to the point (0,0,0) in our world, looking at our target which will be located at our positionVector variable, and our world's up direction is (0,1,0). It's usually safe to assume the up direction of the world is (0,1,0). You might ask, how did we setup the view frustum before we had the camera location and target direction? Well we never specified where to place the frustum, we only told the computer what the frustum's definitions are, our camera will be the location of the view point and the frustum will branch out from that point. It helps to think of the camera view as the actual point the camera is at and the projection as what is shown, or projected, onto the screen.

## MESHES AND EFFECTS

Next our Render() method contains two foreach loops which both set values and draw each mesh of our model. A mesh is the plane area between vertices on a model. Our model already contains a property with the number of meshes defined and we loop through each one using this line:

```
foreach (ModelMesh mesh in gModel.Meshes)
```

gModel is our global class model which we have filled in the constructor with the model we'd like to use. Each mesh itself contains an effect which is applied.

**Note:** There is loads of information on the internet which I won't go into in this specific tutorial but will build up to later on. If you're interested right now search the internet for vertex shaders, pixel shaders, XNA effects, or a combination of those. (I've also written multiple works on pixel and vertex shaders, two of which you can find here:

<http://www.insidegamer.org/documents/Understanding%20Pixel%20And%20Vertex%20Shaders.pdf> and here: <http://www.insidegamer.org/documents/Effects%20of%20Shader%20Technology.pdf> ).

Simply put, effects define lighting and transformations applied to our view or the way our scene is projected onto the screen. As a quick background, effects used to be built into the hardware graphics processing unit (or graphics card) and didn't allow games to really set themselves apart until programmable effects and graphics processing units came out. Even though these effects were finally editable by the programmer, there was still a default effect style which is called the fixed pipeline. Conversely, the programmable shader effects were created and sent through what was called the programmable pipeline. For years DirectX had both options, fixed pipeline and programmable pipeline. Now, XNA has completely removed the fixed pipeline in favor of the programmable pipeline. Instead, a class has been defined that emulates what the fixed pipeline provided, that class is the BasicEffect class. In later tutorials we'll create our own effects and shaders but for now we'll stick with BasicEffect.

Back to the code, our second foreach runs through each effect that is run on the current mesh defined by our above loop. We run this using the following code:

```
foreach (BasicEffect effect in mesh.Effects)
```

Remember, our purpose at this point is to define the way every effect looks on every mesh to properly render them and display them on the screen. As kind of an added touch we call EnableDefaultLighting() from the current effect to show a little light in the scene which will create some brightness or darkness on different sides of the model.

```
//Run default lighting  
effect.EnableDefaultLighting();
```

This method is optional and without it you'll get an ambient light that will shine the same color and brightness on every surface. Next, we set the view and projection matrices to our respective effect properties which,

again, will be used to display the effect and in turn the mesh. The final line of this loop sets what is called the world matrix property of our effect.

```
//Model Matrices
effect.View = view;
effect.Projection = projection;
effect.World = transforms[mesh.ParentBone.Index] *
    Matrix.CreateTranslation(positionVector);
```

So far we've defined two matrices used for rendering, the view matrix (camera position and target) and the projection matrix (view frustum distance and settings). There is one more traditional matrix in rendering and that's the world matrix. We've defined everything so far in 3D coordinates with an X-, Y-, and Z-axis. If you haven't noticed, your computer monitor does not project in 3D, it projects a flat 2D screen. Suppose your monitor projected in 3D, you would be able to move your mouse left, right, and forward and back but we currently don't have that. The purpose of the world matrix is to take those 3D positions and compress it down to 2D. Imagine taking a paper cup and dropping a heavy book on top of it and making it completely flat, then taking off the book and looking at the cup from the top. Back to code, we set the world property as such:

```
effect.World = transforms[mesh.ParentBone.Index] *
    Matrix.CreateTranslation(positionVector);
```

Again, the call to the transforms matrix is only necessary if the model is complex and has a heirarchal structure. Although, if you do have a model like this the transforms matrix will contain potential shaping (such as rotations and scales) and will effect the way your effect and mesh need to be translated to 2D. We then multiply it by a translation matrix which is created using the positionVector and tells the game where our model should be placed. If you want to alter the way the model appears or is translated into 2D space, you can add rotations and scales in this line using "Matrix." and following it with which operation you'd like (CreateRotationY(), CreateScale(), etc...). Just make sure you multiply your rotation or scale with the translation matrix and transform matrix.

```
//Draw the mesh
mesh.Draw();
```

Finally, the last line in the Render() method is done outside the effects loop but within the mesh loop. We call the Draw() method of the mesh to draw the mesh to the screen using the parameters we defined in the effects loop (in this case we didn't have any direct changes to the mesh in the mesh loop but it would take any of those changes as well).

## IMPLEMENTING THE GAMEMODEL CLASS

The hard part is over, the GameModel class is finished and all we need to do is go back to our main game file (Game1.cs) and implement the class. First let's create a GameModel object stemming from our class. Near the top of your Game1 class you should see two lines creating your GraphicsDeviceManager and ContentManager objects. Just under this add this line:

```
GameModel myHouse;
```

There are plenty of free models on the internet or you can make your own if you have any type of 3D model program, just make sure your model has the extension of .x or .fbx (these are the only two formats supported by XNA). You can find a simple house model I've created here:

<http://www.insidegamer.org/documents/houseModel.zip>. Make sure you include your model and texture files into your project by right clicking on the Content folder, selecting 'Add Existing Item...', and browsing to the files. Visual Studio will create copies of these files and place them in your solution's directory.



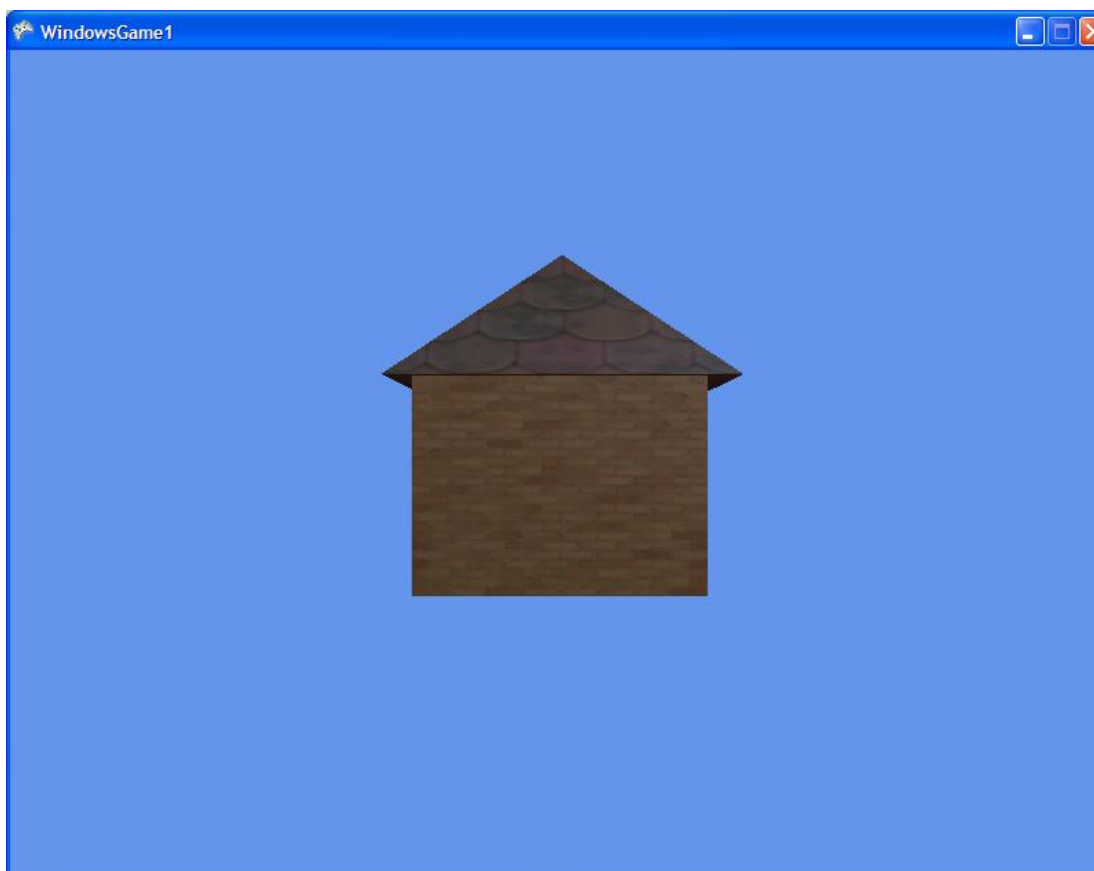
Next we need to call the constructor of the class object which we can do in the overridden Initialize() method XNA creates with a new project. You should see a comment within this method which you should add the following line under (I've added the comment line for better understanding):

```
// TODO: Add your initialization logic here
myHouse = new GameModel(this.Content, graphics, "houseModel");
```

Using the 'new' keyword a new instantiation of the class is created and passed the three objects we asked for (our game's ContentManager, GraphicsDeviceManager, and model name). Notice the model doesn't need the extension attached, just the name of the file. Our class at this point in the code will have our model created, our aspect ratio defined, and set our model to a default position in the world. We've set our camera at this point to that same position so we'll want to move the model using the SetPosition() method we've defined. Also, render it just after we place it so we can do both these operations in the overridden Draw() method Game1 contains. Place this code within your Draw() (you should see the comment below already inside your Draw()):

```
myHouse.SetPosition(0.0f, 0.0f, 5.0f);
// TODO: Add your drawing code here
myHouse.Render();
```

Work with adjusting these values depending on the size of your model, the more you test the better you will understand the model you're using and it's positioning. Notice, the model I've created has it's center at the origin. If the model sat on the origin, the camera would either always look below the model when set to (0.0f, 0.0f, 0.0f) or always above it if the model was moved down. Take a second to look back at your GameModel class and your Game1 class. All the code listed in your GameModel class can be placed directly in your Game1 class but would get really messy. Imagine having four models you want to put in the scene, you would need to rewrite the render code 4 times in your Draw() method (I've tried it with only two models and it was extremely messy!).



## CONCLUSION

As simplistic as it may seem (or may not seem) the GameModel class is a massive step forward. We skipped putting all this directly in Game1 because game programming is really based on dynamic code. If you have 10 levels, you don't want to write GameModel classes for each level, you want one GameModel class that brings in whatever models are needed (the models are defined in each level, not the model import class). If you've noticed, there's a problem with this class. Games are filled with hundreds of models and we had to code an object to this class one-by-one for each model we wanted. A more advanced and standard way is to have a class that manages all the models and creates them dynamically (something we'll get into much later as we build onto this class). The next problem we have is that we can't move our camera which will be fixed in the next tutorial where we build a camera class and integrate it with our models. Don't worry if you don't completely understand the view, projection, and world matrices (it took me months to completely understand the difference between them and what they do). If you aren't sure you could tell another programmer in words what each piece of this class does, just keep reading it over or even research the topic online until you can.

## SUGGESTED EXERCISES

- 1) Create a second instantiation of the GameModel class and move the objects so they sit side by side.
- 2) Rotate the model about the different axis.
- 3) Create a method in the GameModel class where you can specify the rotation of the model so you can change it in your main file. Use this new method to constantly rotate the model while the game is running.
- 4) Create a method in the same manner as Problem 3 but allow this method to scale the object.
- 5) Use some basic input to rotate a model. [HINT: Look at the default input check XNA builds and use that with IntelliSense to rotate the model]