XNA Development: Tutorial 6

By Matthew Christian (Matt@InsideGamer.org)

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

One of the most important portions of a video game isn't even that apparent. Game states are used to describe what 'state' the game is currently in. One of the greatest example is the pause menu. When playing a game, pressing pause prevents the game from continuing on while you are at that screen (assuming the game is single player, not multiplayer). But if you think carefully, many things happen as the game goes from the 'in-game' state to the 'paused' state. The controls have changed, allowing you to move through the pause menu and not move in the game. Maybe some audio has softly begun to play while you navigate the pause menu. A great way to use game states is by creating a class that holds a value defining what state the game is in.

GameState Class Design

While there isn't much 'theory' as it comes to game states, it's hard to imagine all that goes into changing states before actually writing your own game state 'manager' (this tutorial refers to game state manager sometimes and, in the scope of this writing, will refer to the GameState class). The biggest advantage to a game state manager is structuring the code around it. Using the paused game example mentioned above, the pause menu should only show when the game is paused right? And the game should only allow the player to move his/her character when they're at the 'in-game' state, correct? Using a global game state manager as well as some conditional loops (if loops primarily) we can tell the game when to execute certain code.

The game state class has some very basic methods including the constructor and destructor, and some methods to work with the current state. Notice in Diagram 1 there are also three items without parameter lists (parenthesis). InGame, Loading, and Paused aren't functions, they are called properties. Properties are typically used to define the object they are built into (remember, we instantiate an object of the GameState class just like we did with the GameModel, GameCamera, and GameSprite classes). Here's a better example, typical properties of a simple text box could be size, background color, foreground color, or anything that describes the look or (sometimes) action of the object. More will be explained about these properties later.

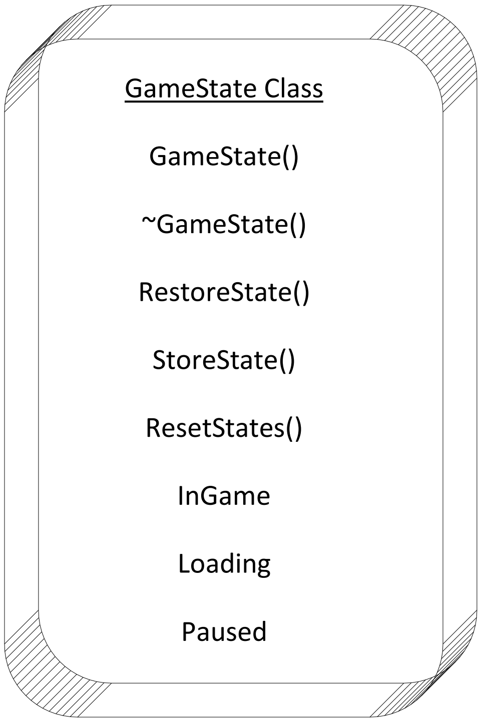


Diagram 1 –GameState Class

Programming the GameState Class

As always, create a new class file named GameState.cs and add the following in place of the `using` statements at the top:

```
#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
```

First comes the global class variables. Place these within your class brackets:

```
    //Bools defining state
    private bool _InGame;
    private bool _Paused;
    private bool _Loading;

    //Temp string to store state
    private String _stateStore;
```

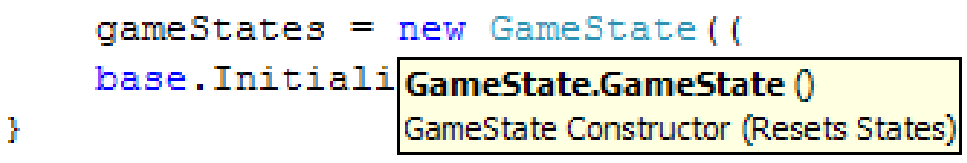
The first three boolean variables are going to act as the 3 possible states our game is in. Each is pretty self explanatory; _InGame is used when in the actual game, _Paused is used when the game is paused, and _Loading is used when the game is loading. These variables have been written differently since each begins with an underscore, and the reasoning is simple. Later, when the properties are coded we want to call those 'InGame', 'Paused', and 'Loading' respectively so they make sense but if we do that we can't have variables named the same way. So, the variables are precedded with an underscore. (Typically, I'll start variables with underscores only if I'm using them in a property. It's not great programming to even use them like this in a case like this but it helps the variable stand out as a property variable.) Second, the _stateStore string variable is created. While it does begin with an underscore, it isn't a property variable. It's simply a member variable used to store the current state before it is written over. More on this variable will come later.

GameState()

The GameState() constructor is very simple, it calls one of our other methods, ResetStates(), to make sure no state is currently set:

```
    /// <summary>
    /// GameState Constructor (Resets States)
    /// </summary>
    public GameState()
    {
        ResetStates();
    }
```

You'll notice the '///<summary>' tag has appeared. If you press the forward slash character 3 times in Visual Studio above a function or variable, this little section appears and works as a comment. But, when you access this method in a different file (like we will later in Game1.cs), the summary section will appear in the yellow Intellisense Tooltip which is a huge help if you have a large project!



```
gameStates = new GameState((
base.Initiali
}
```

GameState.GameState ()
GameState Constructor (Resets States)

Figure 1 - Intellisense Tooltip using Summary

~GameState()

The GameState destructor is empty and has no real clean up duties but we'll add it to be consistant throughout the project:

```
    /// <summary>
```

```
/// GameState Destructor
/// </summary>
~GameState()
{

}
```

ResetStates()

ResetStates() is one of the, if not the most, crucial method to the GameState class. When moving from a state, such as in-game, to a state, such as paused, the game should make sure to set the 'In-Game' state to false and set the 'Paused' state to true. Since the game is being paused, there should only be one true state and all the others should be false (you should **never** have a point in the game where the system is experiencing 2 states simultaniously on the same game). To simplify this, the ResetStates() method will reset every state we've created to false. The code follows:

```
/// <summary>
/// Resets all states
/// </summary>
public void ResetStates()
{
    InGame = false;
    Paused = false;
    Loading = false;
}
```

One of the downfalls of this method occurs as you get more states within your game. For each state you add, you'll need to add a corresponding line to this method.

StoreState()

There could be times in the game where the system will go to a certain state and need to come back to the previous state. Here's a great example; imagine playing a game and you're at the pause menu and want to quit the game. When you click 'Quit Game' a dialogue box pops up and says 'Are you sure you want to quit?' allowing you to click 'Yes' to quit or 'No' to return to the pause menu. Little did you know it at the time, but when you clicked 'Quit Game' the state was stored, set to 'DialogueBox', and when you clicked 'No' the state was restored back to 'Paused'. Here is the code for the StoreState() method:

```
/// <summary>
/// Stores currently set state
/// </summary>
public void StoreState()
{
    if (InGame)
        _stateStore = "InGame";

    if (Paused)
        _stateStore = "Paused";

    if (Loading)
        _stateStore = "Loading";
}
```

The method checks to see which state is currently set to true (remember, only 1 state will be true at any given time). If the state is true we set the storing variable to a string of the state name. As with the ResetStates() method, the greatest downfall is having to add more lines of code for each state.

RestoreState()

Just above, the StoreState() method held the current state and allowed another state to be active while remembering what to go back to. The portion of that which is currently missing is restoring the state back to what it was and can be done like so:

```
/// <summary>
/// Restores currently saved state
/// </summary>
public void RestoreState()
{
    ResetStates();

    if (_stateStore == "InGame")
        _InGame = true;

    if (_stateStore == "Paused")
        _Paused = true;

    if (_stateStore == "Loading")
        _Loading = true;
}
```

First, ResetStates() is called to make sure we're working with completely blank states. Then, we check the string stored in _stateStore against a hardcoded string to see if the two match. If they do match, we set that state to true. Many parts of this code can be considered messy for a multitude of reasons. First of them, yet again we would need more lines of code as new states are created. Secondly, we're doing a version of string checking that could be considered dangerous. If you store the state as 'PauseMenu' and try to restore it as 'Paused' it won't work. For the purposes of this writing, this method of restoring will work and it should work for, up to, medium-sized projects. Remember to keep straight which strings you store so you can return them correctly (state strings would be good listed in a technical design document).

Properties

This whole time we've talked about setting and resetting states but haven't actually seen how we'll set the 'underscore variables' declared above. Properties can have one or two 'sub-methods', or a 'get' and a 'set'. There are many different types of properties but the two most common ways you'll see properties defined will be as a readable, non-writable property or a readable and writable property. For example, the default text in a text box can be accessed by

```
if ( myTextBox.Text == "Loading")...
```

Or can be set by

```
myTextBox.Text = "Hello";
```

The 'Text Property' is an example of a readable and writable property. Readable and non-writable properties can be accessed the first way the 'Text' property of myTextBox was, but cannot be set like in the second part. We want our game to be able to change the state so we'll set the properties to allow reading and writing:

```
/// <summary>
/// Get or set to true creates In-Game state
/// </summary>
public bool InGame
{
    get { return _InGame; }
    set { _InGame = value; }
```

```
        }

        /// <summary>
        /// Get or set to true creates Loading state
        /// </summary>
        public bool Loading
        {
            get { return _Loading; }
            set { _Loading = value; }
        }

        /// <summary>
        /// Get or set to true creates Paused state
        /// </summary>
        public bool Paused
        {
            get { return _Paused; }
            set { _Paused = value; }
        }
```

Get is a predefined procedure used to specify the 'readable' part of our property and set is used to allow the property to be 'writable'. Obviously, when the property is read it returns the value and when it is set brings in a programmed value with the 'value' keyword.

Implementation

Before going right into the implementation of the GameState class, you should understand what is going to be added in the next portion of the tutorial. Added in the next pages is a very basic pause menu that will be activated with the spacebar and will allow the player to click 'Exit Game' using mouse input and will either exit (if they select exit) or return to the game if they press spacebar again. Previously, the classes built to create models and sprites were easily implemented after they were built. Implementing the GameState class is just as easy but actually showing results is the messy part. We could simply create a new object and flip the game state every once in awhile but that would be useless unless something in the game changed that utilized a state change. What I'm trying to say is, we need to add code separate of this class and it's implementation to display, check input, and hide the pause menu so it may seem like there is more work here, but it'll pay off. Also, the code coming up is slightly towards the more advanced side but there should be ample explanation of everything. If you don't understand something, reread it slower and try to work it out on paper to see what's happening.

Updating the GameSprite Class

The GameSprite class has some considerable changes including new methods and updates to every update method contained in the class.

isVisible

Open up your GameSprite.cs class file and locate the sprite struct (should be located at the top). We're going to add a boolean variable to this struct to determine if we want to draw the sprite. The full sprite struct should resemble this:

```
    //New sprite Struct
    public struct sprite
    {
        public string resourceName;
        public Vector2 screenPos;
        public Texture2D spriteTex;
        public float rotation;
        public float scale;
        public bool isVisible;
    };
```

Notice the final variable is now a boolean called isVisible. Why would we need this variable? Well, when that user opens the pause menu and places the cursor over the 'Exit Game' sprite, we want it to be highlighted to signify that item is being hovered over. It would be incredibly hard if not impossible to generate a stroke around the font so we add two sprites when the pause menu is loaded, let's call them 'ExitGame' and 'ExitGameHovered'. When the mouse moves over (or hovers) the 'ExitGame' sprite, we quickly set it to invisible and set 'ExitGameHovered' to visible. This method is quicker than replacing the whole sprite every time although it uses more space by storing that invisible sprite in the sprite list. If you remember one thing from this section, remember that there are two separate sprites, one is always hidden, and they swap depending on if the mouse is hovering over the sprite or not.

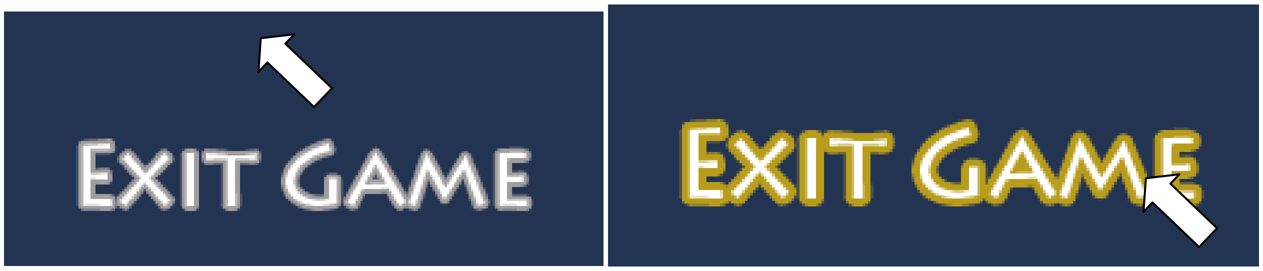


Figure 2 - No Outline (not hovered; left), Outline (hovered; right)

Updating the Updates

Now that isVisible is added to the sprite struct it needs to be set in the AddSprite() method and all the UpdateSprite() methods we have (that includes UpdateSpriteRotation(), UpdateSpriteScale()). I won't go into too much detail about updating those because there are really only 2-3 changes per method. Follow these instructions to update them:

- Add a new parameter, 'bool visible' to the AddSprite() and UpdateSprite() methods' parameter list
- When declaring the newSprite object, add:

```
newSprite.isVisible = visible;
```

- For UpdateSpriteRotation() and UpdateSpriteScale(), add:

```
newSprite.isVisible = oldSprite.isVisible;
```

UpdateSpriteVisibility()

The whole purpose of isVisible is to be able to set the sprite as 'invisible' from outside the class. UpdateSpriteVisibility() will be our helper method to access that variable.

```
/// <summary>
/// Update sprite visibility based on handle
/// </summary>
/// <param name="handle"></param>
/// <param name="spriteName"></param>
/// <param name="scale"></param>
public void UpdateSpriteVisibility(string handle, string spriteName, bool
       isVisible)
{
    //Default variables
```

```
        int position = 0;
        bool found = false;

        //Find sprite
        for (int i = 0; i < gameSprites.Count; i++)
        {
            if (gameSprites[i].resourceName == handle)
            {
                //Found sprite!
                found = true;
                position = i;
            }
        }

        //Can't swap if we don't have an old one
        if (found)
        {
            //Create a new sprite and old holder
            sprite newSprite = new sprite();
            sprite oldSprite = new sprite();

            //Store old sprite
            oldSprite = gameSprites[position];

            //Fill out new sprite
            newSprite.resourceName = handle;
            newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" +
                 spriteName);
            newSprite.screenPos.X = oldSprite.screenPos.X;
            newSprite.screenPos.Y = oldSprite.screenPos.Y;
            newSprite.rotation = oldSprite.rotation;
            newSprite.scale = oldSprite.scale;
            newSprite.isVisible = isVisible;

            //Swap in new sprite
            gameSprites[position] = newSprite;
        }
    }
```

This code should look pretty familiar considering it's basically an exact copy of UpdateSpriteRotation() and/or UpdateSpriteScale(). The only difference is that we're now modifying the isVisible variable instead of the rotation or scale. If you need a refresher, all that is done is the list is searched until a matching sprite is found, then that sprite is swapped out with a new sprite set to all the old values except with a new visible value.

IsSpriteVisible()

We just added the method to modify the isVisible variable, now it's time to access that variable's value via IsSpriteVisible().

```
    /// <summary>
    /// Returns a bool; true if visible
    /// </summary>
    /// <param name="handle"></param>
    /// <returns></returns>
    public bool IsSpriteVisible(string handle)
    {
        //Default variables
        int position = 0;

        //Look for sprite in list
        for (int i = 0; i < gameSprites.Count; i++)
        {
            if (gameSprites[i].resourceName == handle)
            {
                //Found sprite!
                position = i;
            }
        }

        return gameSprites[position].isVisible;
    }
```

Again, pretty simple code. The list is searched through for the matching sprite and the value of isVisible is returned.

GetBoundingBox()

So we have the means to determine if a sprite is visible and set/change its visibility. Now to determine when that is done, we need to know when the mouse is hovering over the sprite. A bounding box is the 'wrapper' around a sprite or model that determines things such as collision. For example, if you have a basketball and press on it, you are pushing on the outer edge of the ball and it will move. The bounding box of the basketball can be considered the outermost layer of the ball, the layer which you are pushing against. Your hand against the ball is considered a 'collision'. Suffice it to say, the bounding box of an object is it's outermost layer. What we'll check is not if the mouse is pushing against the 'Exit Game' sprite (that doesn't really make any sense!), but we'll check if the mouse location is within the bounding box. Get ready for some heavy code:

```
    /// <summary>
    /// Returns Rectangle shaped Bounding Box
    /// </summary>
    /// <param name="handle"></param>
    /// <returns></returns>
    public Rectangle GetBoundingBox(string handle)
    {
        //Default variables
        int position = 0;

        //Look for sprite in list
        for (int i = 0; i < gameSprites.Count; i++)
        {
            if (gameSprites[i].resourceName == handle)
            {
                //Found sprite!
                position = i;
            }
        }

        Rectangle rect = new Rectangle((int)(gameSprites[position].screenPos.X -
            (gameSprites[position].spriteTex.Width / 2 *
           gameSprites[position].scale)),
            (int)(gameSprites[position].screenPos.Y -
            (gameSprites[position].spriteTex.Height / 2 *
           gameSprites[position].scale)),
            gameSprites[position].spriteTex.Width * (int)gameSprites[position].scale,
            gameSprites[position].spriteTex.Height *
            (int)gameSprites[position].scale);
        return rect;
    }
```

The method begins easily with the function declaration and to search for the sprite we are looking for, so far so good. Then, suddenly comes the nastiest piece of code you've seen in this tutorial series. I'll break it down piece by piece and hopefully you'll easily understand the final product. **Note:** If you type that code out, please be careful for new lines. The formatting of this document forces single lines on multiple lines and it's not suggested it's retyped that way. For your reference, the first line should be everything through the third line of the Rectangle statement, the second line should be lines 4-6, the third line should be line 7, and the final should be lines 8-9.

First we create a new variable of type 'Rectangle' and use its constructor to make a new rectangle object. According to MSDN, the Rectangle constructor is as follows:

```
public Rectangle (
         int x,
         int y,
         int width,
         int height
)
```

The X and Y variable of that define the upper left corner of our rectangle area. Width and Height define the width and height of the rectangle respectively. These four items are the simplest way to define a rectangle. If you want to experiment, take a piece of paper and a pencil and draw a large dot somewhere in the center of the page. Then, draw a line 2 inches to the right, and 1 inch down, and you've successfully reached the furthest point in the rectangle. The same operation can be done with simply drawing down first and then to the right and you've got your rectangle.

To help keep track of where we are, let's use pseudocode to understand what we've done. Pseudocode, for anyone who doesn't know, is like writing a program but generally stating what happens instead of actually writing real code. So at this point we've got this:

```
Rectangle rect = new Rectangle(x, y, width, height);
```

Now we set the X value to:

```
((int)(gameSprites[position].screenPos.X - (gameSprites[position].spriteTex.Width /
      2 * gameSprites[position].scale))
```

Forget the (int) portion for now, and look at the rest of it. First, it takes the sprites on-screen X-positon and subtracts from it a value created by grabbing the sprite width divided by two, and multiplying it by the scale (if the scale is different, the bounding box should be smaller or bigger respectively). Considering scale is typically 1.0, we've essentially taken the X-position of the sprite and moved it to the left edge of the sprite. Since Rectangle only allows integer X amounts and our scale is a float, we need to 'cast' it as an integer. For example, if you cast 3.0 as an integer, your answer would be 3. That is the reason for the (int) item in the front. Now we know the left side of the sprite that's our X-coordinate! One parameter down, three to go.

Second, we find the Y-coordinate of the top of the box (remember, we need the upper left corner).

```
(int)(gameSprites[position].screenPos.Y - (gameSprites[position].spriteTex.Height /
      2 * gameSprites[position].scale))
```

If you understood the X-coordinate, you should understand this as it's practically the same. Notice, however, that we subtract half the height to get the top (remember, screen coordinates have positive X to the right and positive Y towards the bottom of the screen, traditionally). Following our pseudocode we now can say we have something like this:

```
Rectangle rect = new Rectangle(KNOWN_x, KNOWN_y, width, height);
```

The last two parameters are the width and the height of the sprite which we've actually used before in our render method! The Texture2D item in our sprite struct that holds the actual sprite contains access to the file's height and width. Thankfully these are easily accessed with the width being:

```
gameSprites[position].spriteTex.Width * (int)gameSprites[position].scale
```

And the height being:

```
gameSprites[position].spriteTex.Height * (int)gameSprites[position].scale)
```

Notice they are both multiplied by the scale of the sprite to account for any scale distortion made by scaling it. Our Rectangle is now complete, containing the upper-left corner's (X,Y) coordinates and the width and height of the sprite. All that is done then, is the rectangle is returned by the method. This is an immensely useful method and can be used for collision in 2D games as well so make sure you understand what it does.

DrawSprites()

Being added to the DrawSprites() method is a simple IF statement around the Draw() to only draw the sprite if it's isVisible item is set to true:

```
/// <summary>
/// Draw all sprites
/// </summary>
public void DrawSprites()
{
    //Begin our gameBatch using properties
    gameBatch.Begin(SpriteBlendMode.AlphaBlend, SpriteSortMode.Deferred,
        SaveStateMode.SaveState);

    for (int i = 0; i < gameSprites.Count; i++)
    {
        if (gameSprites[i].isVisible)
        {
            //Draw each item on the list
            gameBatch.Draw(gameSprites[i].spriteTex, gameSprites[i].screenPos,
                null, Color.White, gameSprites[i].rotation, new
                Vector2(gameSprites[i].spriteTex.Width / 2,
                gameSprites[i].spriteTex.Height / 2), gameSprites[i].scale,
                SpriteEffects.None, 0.0f);
        }
    }

    //Finish game batch code, end
    gameBatch.End();
}
```

Implementing the GameState Class

As always, we'll create a new object of the new class type (the new class being GameState). Add the following below your current list of global object variables:

```
GameState gameStates;
```

Now, we also need to create a new structure to hold what key is currently pressed, or what was pressed last. Following your global variables add this code:

```
public struct keylog
{
    public Keys _key;
```

```
        public bool _keyPressed;
    };

    keylog remKey;
```

Previously, we've worked with handling user input with checking items such as moving the camera forward or back. In those cases, we would check the user input each game loop and if it was pressed down would perform the respective action. Now, imagine that but with a 'Pause' button. The user presses pause and the game flips into the Pause state and displays the pause screen. But, because game loops are so incredibly fast, the game will come back into that loop, recognize the player still has the pause button held down, and flip out of the Pause state and back into the game. What we want to do is have the pause state initialize, and not to access the pause button until it's unpressed and pressed again.

This is where the keylog struct comes into play. Keylog will store the current pressed button and if the key is pressed. Then, when we check to see if the pause button is down, we can check and see if keylog has anything stored, if it does we don't want to flip out of the pause state. If keylog is blank, nothing critical is currently down and we can process whatever needs to run. **Note:** When I say 'critical', I'm referring to crucial keys like the pause button. We don't care if the player is holding forward when they go into the pause state, we only care about the pause button in that case.

Now, instantiate your new gameStates variable and set your keylog to some default values by placing the following code in your Initialize() method of the Game1.cs file.

```
        gameStates = new GameState();

        remKey._key = Keys.None;
        remKey._keyPressed = false;
```

LoadContent() Updates

There are a few small updates to the LoadContent() method of the Game1 class. Previously, we used this section to initialize our GameSprite object and load all our sprite items. This section will continue to do the same but since we updated the AddSprite() methods to include the isVisible variable, we need to update it. Here's the full LoadContent() method:

```
    protected override void LoadContent()
    {
        gameStates.ResetStates();
        gameStates.Loading = true;

        // Create a new SpriteBatch, which can be used to draw textures.
        spriteBatch = new SpriteBatch(GraphicsDevice);
        gameSprites = new GameSprite(this.Content, spriteBatch);

        gameSprites.AddSprite("RadarBG", "RadarBG",
              graphics.GraphicsDevice.Viewport.Width - 100, 100, 0.0f, 0.7f, true);

        gameSprites.AddSprite("RadarChar", "RadarChar",
              graphics.GraphicsDevice.Viewport.Width - 100, 100, 0.0f, 0.7f, true);

        gameSprites.AddSprite("4Buttons", "4Buttons",
              graphics.GraphicsDevice.Viewport.Width - 90,
              graphics.GraphicsDevice.Viewport.Height - 90, 0.0f, 1.0f, true);

        gameSprites.AddSprite("HealthBG", "HealthBG", 140,
              graphics.GraphicsDevice.Viewport.Height - 100, 0.0f, 1.0f, true);
```

```
            gameSprites.AddSprite("HealthBorder", "HealthBorder", 140,
                  graphics.GraphicsDevice.Viewport.Height - 100, 0.0f, 1.0f, true);

            //Load Pause Screen
            gameSprites.AddSprite("PauseBG", "PauseBG",
                  graphics.GraphicsDevice.Viewport.Width/2,
                  graphics.GraphicsDevice.Viewport.Height/2, 0.0f, 1.0f, false);

            gameSprites.AddSprite("PauseMenuLargeBG", "PauseMenuLargeBG",
                  graphics.GraphicsDevice.Viewport.Width/2,
                  graphics.GraphicsDevice.Viewport.Height/2, 0.0f, 1.0f, false);

            gameSprites.AddSprite("PauseMenuExit", "PauseMenuExit",
                  graphics.GraphicsDevice.Viewport.Width / 2,
                  graphics.GraphicsDevice.Viewport.Height / 2 + 200, 0.0f, 1.0f, false);

            gameSprites.AddSprite("PauseMenuExitSelected", "PauseMenuExitSelected",
                  graphics.GraphicsDevice.Viewport.Width / 2,
                  graphics.GraphicsDevice.Viewport.Height / 2 + 200, 0.0f, 1.0f, false);

            gameStates.ResetStates();
            gameStates.InGame = true;
        }
```

At the top of this method, the current state of the game is set to Loading. One thing you need to notice is the way the state is set. Since we don't want two states at once, we need to call gameStates.ResetStates() to set everything to false followed by immediately setting a certain state to true. Make **sure** to call ResetStates() before each change and always to set immediately after it.

Next, each of the previous AddSprite() calls we had has the argument 'true' sent at the end which tells GameSprite that these sprites should be displayed, not hidden. We're starting the game and going right into the InGame state after this, if we had a main menu we'd load that here instead and show that.

Since this is a small project and uses a relatively small number of content items, we can load all the sprites for the pause menu right now and just hide them until they're needed. Of course, if you had a large project you would probably want to load the sprites when the pause menu was initiated (notice how some games have a quick Loading Pause Menu state when the player presses pause). They are all basic AddSprite() method calls with the only difference being the position of them and the visibility set to false. Finally we set the game state to InGame and are ready to go 'In Game'.

Note: You can find these sprites at http://www.insidegamer.org/documents/pauseSprites.zip. Add them to your project by right clicking on the Content subfolder and selecting 'Add Existing Item…'.

Updating Update()

The majority of the Update() method found in the Game1.cs file changes now due to the implementation of the GameState class. First, the code:

```
        protected override void Update(GameTime gameTime)
        {
            if (Keyboard.GetState().IsKeyUp(Keys.Space) && (remKey._key == Keys.Space &&
                remKey._keyPressed == true))
            {
                remKey._key = Keys.None;
                remKey._keyPressed = false;
            }

            //Exit game
```

```
if (Keyboard.GetState().IsKeyDown(Keys.Escape))
    this.Exit();

if (gameStates.InGame)
{
    //Move forward
    if (Keyboard.GetState().IsKeyDown(Keys.W))
        camera.MoveForward();

    //Move Back
    if (Keyboard.GetState().IsKeyDown(Keys.S))
        camera.MoveBack();

    //Strafe Left
    if (Keyboard.GetState().IsKeyDown(Keys.A))
        camera.StrafeLeft();

    //Strafe Right
    if (Keyboard.GetState().IsKeyDown(Keys.D))
        camera.StrafeRight();

    //Turn left
    if (Keyboard.GetState().IsKeyDown(Keys.Left))
    {
        camera.turnAmt += camera.rotationSpeed;
        camera.RotateCamera();
    }

    //Turn right
    if (Keyboard.GetState().IsKeyDown(Keys.Right))
    {
        camera.turnAmt -= camera.rotationSpeed;
        camera.RotateCamera();
    }

    //Pause Game
    if (Keyboard.GetState().IsKeyDown(Keys.Space) && (remKey._key !=
        Keys.Space && remKey._keyPressed == false))
    {
        remKey._key = Keys.Space;
        remKey._keyPressed = true;

        ShowPauseMenu();
        this.IsMouseVisible = true;

        gameStates.ResetStates();
        gameStates.Paused = true;
    }
}

if (gameStates.Paused)
{
    CheckPauseMenu();

    if (Keyboard.GetState().IsKeyDown(Keys.Space) && (remKey._key !=
        Keys.Space && remKey._keyPressed == false))
    {
        remKey._key = Keys.Space;
        remKey._keyPressed = true;

        HidePauseMenu();
        this.IsMouseVisible = false;
```

```
                gameStates.ResetStates();
                gameStates.InGame = true;
            }
        }

        base.Update(gameTime);
    }
```

There are three crucial pieces that have changed in this section. The first appears right inside the top of the method and looks like:

```
    if (Keyboard.GetState().IsKeyUp(Keys.Space) && (remKey._key == Keys.Space &&
        remKey._keyPressed == true))
    {
        remKey._key = Keys.None;
        remKey._keyPressed = false;
    }
```

It's crucial that this code is placed at the beginning of the method, this is the piece of code that will check if the pause button (Space button) has been 'unpressed' and if it needs to be reset. If it has been unpressed and keylog currently has it set as the pressed button, keylog is reset and we are allowed to access any Pause or Unpause items that may occur later in the code. (Notice, keylog acts almost as a key manager with support for only one key).

Secondly, an if statement has been layed around the previous button presses we had (excluding the escape button; leaving this button to escape at any time is useful for debugging purposes). This if statement only allows the buttons to be checked only if the gameState manager currently has the 'InGame' state set. If, for example, we're in a Pause state, we don't need to check to see if the player presses forward and move the camera. Typically, a forward press would navigate through the menu, which can now be added if we check for any other button presses during the Pause state.

```
            if (gameStates.InGame)
            {
                  ...
                //Pause Game
                if (Keyboard.GetState().IsKeyDown(Keys.Space) && (remKey._key !=
                    Keys.Space && remKey._keyPressed == false))
                {
                    remKey._key = Keys.Space;
                    remKey._keyPressed = true;

                    ShowPauseMenu();
                    this.IsMouseVisible = true;

                    gameStates.ResetStates();
                    gameStates.Paused = true;
                }
            }
```

Skipping to the bottom of the if statement, we find a new check to see if the spacebar has been pressed. If Space has been pressed and isn't being held down, we keylog the spacebar, call a method ShowPauseMenu(), show the mouse (our pause menu uses mouse interaction), and finally sets the current state.

That leads us to the final update to this method, yeat another if statement blocking off code depending on the current state of the game. The code:

```
            if (gameStates.Paused)
            {
                CheckPauseMenu();
                if (Keyboard.GetState().IsKeyDown(Keys.Space) && (remKey._key !=
                    Keys.Space && remKey._keyPressed == false))
                {
                    remKey._key = Keys.Space;
                    remKey._keyPressed = true;

                    HidePauseMenu();
                    this.IsMouseVisible = false;

                    gameStates.ResetStates();
                    gameStates.InGame = true;
                }
            }
```

First off, if the game state isn't set to paused we don't need to check the pause menu (obviously), hence the if statement. Next, there is a method called CheckPauseMenu() which you'll learn about in an upcoming section. Finally, we check again to see if paused was pressed. Assuming it was pressed and isn't being held down, we set the keylog to remember space is being pressed, call HidePauseMenu() which you'll learn about a bit later, hide the mouse, and finally set the state to InGame.

ShowPauseMenu()

ShowPauseMenu() will be a public method in our Game1 class that updates the visibility of the pause menu sprites.

```
        /// <summary>
        /// Displays Pause Menu
        /// </summary>
        public void ShowPauseMenu()
        {
            //Show Pause Screen
            gameSprites.UpdateSpriteVisibility("PauseBG", "PauseBG", true);

            gameSprites.UpdateSpriteVisibility("PauseMenuLargeBG", "PauseMenuLargeBG",
                  true);

            gameSprites.UpdateSpriteVisibility("PauseMenuExit", "PauseMenuExit", true);
        }
```

There are 4 core pause sprites we're utilizing but only three get set to visible at this point. There is a dark background, a border image, the exit button, and the exit button hovered over but we don't know if the exit button is being hovered so don't show that right now. As shown above, this method is called just before we set the state to Pause.

HidePauseMenu()

Just like the ShowPauseMenu(), when the game moves from the Pause state to the InGame state, we need the pause menu to hide itself again.

```
        /// <summary>
        /// Hides Pause Menu
        /// </summary>
        public void HidePauseMenu()
        {
            //Hide Pause Screen
            gameSprites.UpdateSpriteVisibility("PauseBG", "PauseBG", false);
            gameSprites.UpdateSpriteVisibility("PauseMenuLargeBG", "PauseMenuLargeBG",
                  false);

            gameSprites.UpdateSpriteVisibility("PauseMenuExit", "PauseMenuExit", false);

            gameSprites.UpdateSpriteVisibility("PauseMenuExitSelected",
                  "PauseMenuExitSelected", false);
        }
```

In case the user is hovering the exit button and unpauses the game, we hide the selected exit sprite as well. Again, all that is done here is that the sprites are hidden from view until ShowPauseMenu() is called again.

CheckPauseMenu()

At this point we've implemented ways to show and hide the pause menu as well as sectioned off the code to provide for different states. What's needed now is a way to interact with that exit sprite we're including on our pause menu. As always, the code:

```
        /// <summary>
        /// Checks any pause menu input
        /// </summary>
        public void CheckPauseMenu()
        {
            MouseState currMouse = Mouse.GetState();
            int mouseX = currMouse.X;
            int mouseY = currMouse.Y;

            Rectangle recExit = gameSprites.GetBoundingBox("PauseMenuExit");

            if ((mouseX >= recExit.Left) && (mouseX <= recExit.Right) &&
                (mouseY <= recExit.Bottom) && (mouseY >= recExit.Top))
            {
                if (gameSprites.IsSpriteVisible("PauseMenuExit"))
                {
                    gameSprites.UpdateSpriteVisibility("PauseMenuExit", "PauseMenuExit",
                          false);
                    gameSprites.UpdateSpriteVisibility("PauseMenuExitSelected",
                          "PauseMenuExitSelected", true);
                }

                if (currMouse.LeftButton == ButtonState.Pressed)
                {
                    //End game
                    this.Exit();
                }

            }
            else
            {
                if (gameSprites.IsSpriteVisible("PauseMenuExitSelected"))
                {
                    gameSprites.UpdateSpriteVisibility("PauseMenuExit", "PauseMenuExit",
                  true);
                    gameSprites.UpdateSpriteVisibility("PauseMenuExitSelected",
                  "PauseMenuExitSelected", false);
                }
            }
        }
```

The first three lines of this method create a MouseState variable allowing us access to check the status of the mouse. If you look back at the Update() method, every time we access the keyboard we do something similar to Keyboard.GetState()… This uses the same princple but as Mouse.GetState(). Then, two integer variables are created to store the position of the mouse on the screen. What we'll do is check the position of the mouse on the screen and see if it's passed inside the bounding box of the exit game sprite using the handy GetBoundingBox() method we built into the GameSprite class.

Now, the bounding rectangle is created and set to the sprite box. If the mouse isn't over the box we'll set it to the normal exit sprite and if it's hovering over that area we'll show the hovered sprite. This way, we don't need to check the bounding box of the hovered sprite at all (even though they are the same box).

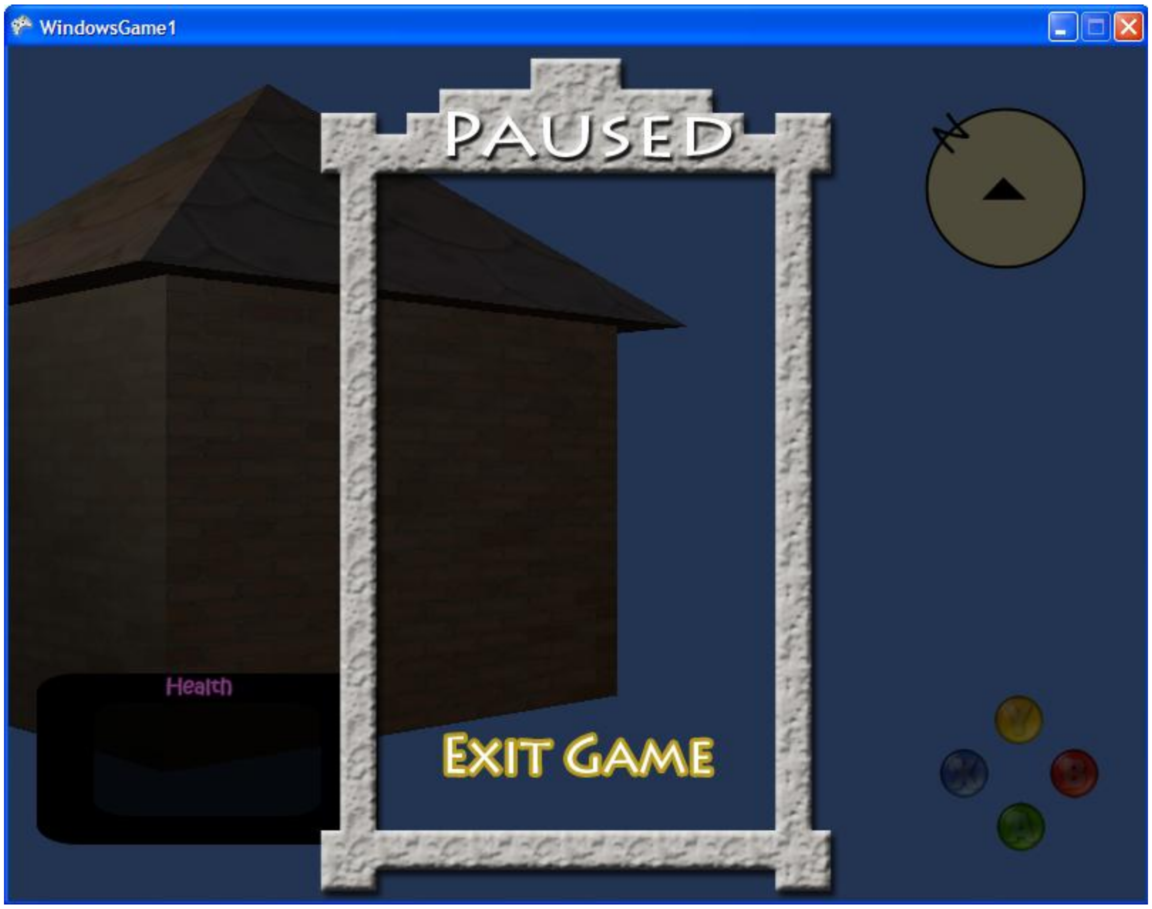
Following the bounding rectangle definition, an if is run to check the mouse position. We check each side to see if it falls on or within the box (we check the position to the right of the left wall, position below the top wall, position above the bottom wall, and position to the left of the right wall). Of course, all these need to be true to confirm the mouse is hovering (if the mouse was on the right edge of the screen, it would always satisfy the left wall check, but won't satisfy any of the others). Let's look at the insides of the if condition:

```
if ((mouseX >= recExit.Left) && (mouseX <= recExit.Right) &&
    (mouseY <= recExit.Bottom) && (mouseY >= recExit.Top))
{
    if (gameSprites.IsSpriteVisible("PauseMenuExit"))
    {
        gameSprites.UpdateSpriteVisibility("PauseMenuExit", "PauseMenuExit",
              false);
        gameSprites.UpdateSpriteVisibility("PauseMenuExitSelected",
              "PauseMenuExitSelected", true);
    }

    if (currMouse.LeftButton == ButtonState.Pressed)
    {
        //End game
        this.Exit();
    }

}
```

So, if the mouse is within the box defined by the edge of the sprite, **and** if the normal exit sprite is showing, hide the normal sprite and display the hovered version. Without that second if, we would be updating the visibility of the sprites every game loop while it was being hovered (remember, game loops occur extremely fast meaning we'd do multiple useless updates every second!). Then, if the left mouse button is clicked while the exit game sprite is being hovered, we exit the game. Of course, there's an else condition in case the mouse isn't within the bounding box:

```
else
{
    if (gameSprites.IsSpriteVisible("PauseMenuExitSelected"))
    {
        gameSprites.UpdateSpriteVisibility("PauseMenuExit", "PauseMenuExit",
      true);
        gameSprites.UpdateSpriteVisibility("PauseMenuExitSelected",
      "PauseMenuExitSelected", false);
    }
}
```

Now, we're not within the bounding box right? Well, if we showed the hover sprite and left, the hover wouldn't go away unless we had some kind of check to determine that the hovered sprite is showing and that the mouse isn't over the sprite anymore. Since the else means we aren't hovering anymore (because the mouse doesn't fall in the box) all we check is if the hovered version of the sprite is showing. If so, we hide it and show the normal sprite. This if is used similar to the one above preventing multiple useless updates every second.



Conclusion

Game states are extremely useful considering the way they are almost unnoticed by the casual gamer. When it comes to programming in different states, it could save you lots of useless running code. The tradeoff of this benefit is that implementing states is tough business. As each state comes into your game, you'll need more and more code to navigate around and through those states. Hopefully you've understood the majority of the code presented in this tutorial because it got pretty dense in some spots. Remember, take your time, understand it all, and you'll have an awesome game demo in no time.

Suggested Exercises

1) Using some type of image software, create more sprites for the pause menu and implement them (including hovered versions if possible).

2) Add a very simple main menu using new sprites and a new game state called MainMenu.

3) Create a menu dialogue prompting when 'Exit Game' is clicked asking 'Are you sure you want to exit?' providing Yes and No as options. Use the StoreState() and RestoreState() methods where needed.

4) Does adding sprites in a specific order matter when it comes to loading pause sprites before in-game sprites? Why or why not?

5) Make a list of other game states that are used in current games. Describe in very general pseudocode how to implement each item.