Contents

Introduction

This tutorial will explain how to use the CS8803AGA game platform. In some places it will assume a basic familiarity with C# and Visual Studio, but will not assume any knowledge of XNA Game Studio. Many features of XNA have been abstracted away, so experienced XNA users may benefit from this tutorial as well (or, they may be better off starting from scratch).

Downloading and Installation

First, download and install Microsoft Visual C# Express 2008 from http://www.microsoft.com/express/Downloads/ (third tab). You can also get 2010, though you may have to go through a couple of prompts converting the solution file (I have had someone verify that it is doable). Next, download and install XNA 3.1 from http://www.microsoft.com/downloads/details.aspx?FamilyID=80782277-d584-42d2-8024-893fcd9d3e82&displaylang=en. Once these are downloaded and installed, open up the "CS8803AGA.sln" file. A solution file consists of a number of projects, where a project contains all of the source code needed to create an exe, static lib, dll, etc.

The five projects within CS8803AGA.sln are the following:

- CS8803AGA: this is an XNA Windows executable; in other words, it contains the code and data packaged into the game's executable and runtime data
- CS8803AGAContentPipeline: normally, XNA auto-generates serializers and deserializers to move between data classes, XML, and binary, but a content pipeline project contains custom versions of these for more fine-tuned control (you can probably get by without ever touching this)
- CS8803AGAEditor: this is a simple Windows forms application which can be used to create, load, and save XML files used to store data (animation metadata, monsters, weapons, etc) for the game
- CS8803AGAGameLibrary: this library contains data types whose instances will be created as XML files and loaded into the game at runtime
- XMLTester: this is a stand-alone application which can be used to ascertain the automatically-generated XML format of a class in the GameLibrary; simply create an instance of the class and run the program and it will output that object into XML

Features

- Implementation of State pattern for gameplay and menus using a stack
- Multithreaded rendering of 2D images and text
- Quad-tree collision detection using rectangles
- Multiplexed input from mouse/keyboard and Xbox360 controllers, as well as "sticks" and "toggles" for ease of use when reading inputs

- Animation playback via sprite sheets and metadata
- Infrastructure for areas, tiling, and decorations
- Editor to create game content in XML, with some support for subclasses (polymorphism)

Trying It Out

Once you have the solution file opened, you're ready to see the game as it stands. Right-click on the CS8803AGA project, Debug->Start New Instance. The game should launch. The arrow keys (or WASD) are used to scroll through menus and move around the game world. Spacebar will cause the character to play an attack animation; if standing to the side of an NPC and you attack twice, the NPC will die. Top and bottom attacks are left as an exercise for the reader. Standing next to an NPC, facing them, and pressing Enter will bring up a "dialogue" page, though in this case that page is an ugly, poorly sized map of the world. Pressing X from the main gameplay state will also bring up a map which can be scrolled upon. Finally, right-clicking will drop a house with the upper-left corner at the mouse position.

Important Tip

If you're new to C# in Visual Studio, the biggest piece of advice that I can give you is the "Resolve" feature for missing classes, structs, enumerations, etc. If you type some class name and it doesn't show up in *light* blue, it means that Visual Studio doesn't recognize it as being available to the file/namespace you're working in. Right click on the name and if Visual Studio can find it anywhere in an assembly which has been referenced, there will be a "Resolve" item in the menu. Put your cursor over it and it will give you the option of adding the appropriate "using" directive to the top of the file automatically. When doing this, make sure you pay attention and always select the "Xna" option if there is a choice, as sometimes there are identically-named classes in System.Drawing which you don't want. Throughout this guide I will not always tell you when you need to add new "using" directives; I will assume you can use the Resolve feature to handle this for you automatically.

Support

Please do your best and work with others to resolve any technical issues you may have. The T-Square forums are a great place to post questions and help others, so please take the time to check them regularly. If you find bugs and manage to fix them please share these with the class.

Explanation of the Content Pipeline

XNA 3.1 supports XML as a major way to author content for a game, and it provides several tools to automate the process of getting XML data loaded at runtime. Here is an outline of the steps that are involved:

- Authoring time
    - A data type is created in the GameLibrary project: let's call it Weapon.cs
    - The content author creates an XML document (either by hand or using some type of editor) which matches the schema of the class; essentially this is an instance of that data type: let's call it FlamingLongswordOfImpendingDoom.xml
    - The XML document is placed somewhere in the Content folder of the main project: let's say Content\Items\Weapons; depending on the type, the user must change some properties in the Properties toolbar so XNA knows how to handle it
- Build time
    - XNA scans that XML document for the class type it represents, and using that type it uses an XML deserializer to turn that XML into an in-memory object (the XML deserializer is generated automatically from the class file unless an alternate is created by hand)
    - The in-memory object is then serialized into an XNA binary format (.xnb) using a binary serializer (again, generated automatically from the class unless an alternate is written by hand)
    - All .xnb files are packaged into some kind of archive that you don't need to worry about, just know that the code can automatically figure out how to extract files from the archive using the paths relative to the Content folder
- Run time
    - Content is loaded through the ContentManager class, which is generally accessed through the Content property of a Game object (in our case, our Game object is an instance of the Engine class)
    - ContentManager.Load<T>(string path) is generally the call used, where T is a type which tells the ContentManager what kind of binary deserializer to use, and the path tells the ContentManager where in the archive to find the .xnb file that you want loaded: in our case, we might have GlobalHelper.getInstance().Engine.Content.Load<Weapon>("Items\Weapons\FlamingLongswordOfImpendingDoom") since the "Content" directory is usually set as the root directory; note the lack of a file extension, XNA figures that out for us
    - Some more information: http://blogs.msdn.com/b/shawnhar/archive/2009/03/25/automatic-xnb-serialization-in-xna-game-studio-3-1.aspx

Tutorial: Game States and Controller Inputs

This tutorial will explain how to read controller (keyboard/mouse or Xbox 360 controller) inputs and manage different states of gameplay. A basic pause screen will be created.

1. Engine states
   a. See "CS8803AGA/engine." Engine states are the various states that the game can be in, such as the main menu, loading a level, in gameplay, at the pause menu, etc. It is maintained as a stack to make it easier to keep track of embedded menu systems and the like. This also allows the drawing of one state on top of another; for instance, a dialogue window may be drawn over the main gameplay screen. Each game frame, the top state on the stack is updated and drawn by Engine; its logic can update or draw other states as needed. The EngineManager is the class which manages these states.
   b. First, create a new state for a pause menu; call it "EngineStatePauseMenu.cs":

```
namespace CS8803AGA.engine
{
    class EngineStatePauseMenu : AEngineState
    {
        public EngineStatePauseMenu(Engine engine)
            : base(engine)
        {
            // nch
        }

        public override void update(Microsoft.Xna.Framework.GameTime gameTime)
        {
            // TODO
        }

        public override void draw()
        {
            // TODO
        }
    }
}
```

2. Input Handling
   a. We now need code to reach the pause state. Ideally, the user should press ESC during gameplay to be taken to the pause screen. Thus, we make the following addition to EngineStateGameplay's update loop. Inputs are checked by querying the devices/InputSet singleton. For keys/buttons, we call "getButton" with an enumerated value representing that key's/button's purpose; true implies that it is pressed and false implies that it is not. (As an aside, the purpose of these enumerated values was to ease the simultaneous development of PC and Xbox360 inputs). In devices/PCControllerInput, we find the mapping which tells us which values map to which keyboard keys. "Esc" is mapped to the CANCEL_BUTTON, so:

```
if (InputSet.getInstance().getButton(InputsEnum.CANCEL_BUTTON))
{
    EngineManager.pushState(new EngineStatePauseMenu(m_engine));
    return;
}
```

b. Feel free to run the code and observe what happens when ESC is pressed.

c. Next we need some logic to get us out of the pause screen, preferably when the user presses “Esc” once again. Furthermore, it would be nice if our pause screen weren’t just a blank screen, so we’ll add code to draw the gameplay state below it on the stack. Make the following changes to EngineStatePauseMenu:

```
public override void update(Microsoft.Xna.Framework.GameTime gameTime)
{
    if (InputSet.getInstance().getButton(InputsEnum.CANCEL_BUTTON))
    {
        EngineManager.popState();
        return;
    }
}

public override void draw()
{
    EngineManager.peekBelowState(this).draw();
}
```

d. Unfortunately, if you try this out, it will yield some very strange behavior. Sometimes, pressing Escape will pause the game correctly and you won’t be able to move around, but other times it won’t. Likewise, if the game is paused, sometimes it will unpause correctly, and other times it won’t. The reason for this is actually simple: when you press ESC, you’re not pressing it for only a single frame. Thus, when in gameplay mode and you press ESC, it switches to the pause menu, but ESC is still pressed, so it switches back, and so on. To prevent problems such as this, InputSet has a feature called “toggles.” When a toggle is turned on for a particular key or button, that key or button will not appear pushed again (virtually) until it has been released for at least one frame (physically). Thus, it is a useful convention to apply toggles to all of the keys when transitioning from one state to another; however, depending on what feels right, you could only apply it to the key used to trigger the transition. Below is an example of how to do this in EngineStatePauseMenu, and you should add a similar call in EngineStateGameplay.

```
using CS8803AGA.devices;
...

if (InputSet.getInstance().getButton(InputsEnum.CANCEL_BUTTON))
{
    InputSet.getInstance().setAllToggles();
    EngineManager.popState();
    return;
}
```

e. The game should now pause and unpause correctly.

Tutorial: Drawing and Menus

1. Drawing
    a. With that problem fixed, let’s make the pause screen look a little nicer. After all, it’s impossible to tell whether the game is currently paused by just looking at it. We should add a visual indicator. Drawing images is done via the GameTexture and DrawCommand classes.
    b. CS8803AGA/rendering/textures/GameTexture is an encapsulation of a loaded image file. It contains a reference to the actual image data (its Texture property) and an array of rectangles (the ImageDimensions property) which delineate separate images within a file which you may want to draw separately (this is useful for sprite sheets, but often ImageDimensions will be a single rectangle of the size of the file). To create a GameTexture, we place an image file in the CS8803AGA/Content project then provide a path to it at runtime (excluding the file extension). For the pause screen, we will use “Content/Sprites/RPG/PopupScreen”; since we want the whole image, we will use the GameTexture constructor with one argument.
    c. In order to actually tell the engine to render something, we use an instance of a CS8803AGA/rendering/textures/DrawCommand. DrawCommand is an encapsulation of an instruction to the engine to render an image – it contains information about position, rotation, scale, color, etc. There are two ways to use DrawCommand. The preferred method is to call DrawBuffer.getInstance().DrawCommands.pushGet(). What this command does is push a DrawCommand into its internal stack which eventually gets sent for rendering, and it returns that object for you to edit its parameters.

d. With that in mind, we make our changes to EngineStatePauseMenu:

```
protected GameTexture m_tPausePage;

public EngineStatePauseMenu(Engine engine)
    : base(engine)
{
    // Load the texture upon creation of the game state
    m_tPausePage = new GameTexture("Sprites/RPG/PopupScreen");
}

...

public override void draw()
{
    EngineManager.peekBelowState(this).draw();

    // Retrieve a DrawCommand; it is automatically sent for drawing
    DrawCommand dc = DrawBuffer.getInstance().DrawCommands.pushGet();

    // Must always set the texture of the command so there is something
    // to draw
    dc.Texture = m_tPausePage;

    // We're not using the Camera, so we want absolute positioning
    // Since it’s a UI element, we would probably use absolute anyway
    dc.CoordinateType = CoordinateTypeEnum.ABSOLUTE;

    // Draw it just above the gameplay depth; this should probably be
    //  a constant stored somewhere else, but I leave data management
    //  decisions to you
    // All draw depths should be between 0 and 1.  See the Constants
    //  class for some examples.
    dc.Depth = Constants.DepthMaxGameplay + 0.001f;

    // Draw to the center of the drawable area
    Point temp = m_engine.GraphicsDevice.Viewport.TitleSafeArea.Center;
    dc.Position = new Vector2(temp.X, temp.Y);

    // And we want it centered on the position that we provided
    dc.Centered = true;
}
```

e. As you can see, there are several ways to alter a DrawCommand. You should familiarize with the arguments that are available and their defaults so that you know which ones to set when drawing an object. Here are some notes:

   i. The current game does not use a Camera object. However, if you wanted scrolling screen areas rather than static ones (e.g. Mario), the Camera would be useful. All DrawCommands with RELATIVE positioning will have their positions adjusted by the Camera’s position at render time, so as long as you keep the Camera in the position you want to display, other world coordinates do not have to change. Set the Camera in GlobalHelper.

   ii. The Destination property will cause rendering to ignore Position and Scale, and instead map the rectangular image into the Destination rectangle. (This has not been thoroughly tested but is provided by XNA so should work.) It is not supported with Camera use but could be made to do so easily by editing the DrawCommand’s draw() function.

f. A warning about multithreading: this platform’s multithreaded rendering does NOT conform to the normal XNA execution loop. This platform has two threads running simultaneously. The first thread is the Update thread in which all of the update() functions of the engine are called. It then calls all of the draw() functions, which as you have seen do not actually draw anything, but place a DrawCommand into a buffer. The second thread, the Draw thread, is concurrently sending DrawCommands from last frame's buffer to the graphics device for rendering. When both threads are finished, the two threads swap stacks of DrawCommands. As such, you should probably never modify the Draw thread unless you really know what you are doing, and you should never try to perform actual rendering from the draw() calls on objects (because they are actually called in the Update thread).

2. Menus
    a. Next we'll create an actual menu for the pause screen. The ui/MenuList class provides a fairly easy way to render text to the screen and change the selection.
    b. Reference the EngineStateMainMenu to see how MenuList can be used. I recommend doing this as an exercise, but if you don't want to, a solution is below.

c.

```
class EngineStatePauseMenu : AEngineState
{
    protected static readonly string[] c_menuItems =
        new string[] { "Return to Game", "Settings", "Quit Game" };

    protected GameTexture m_tPausePage;
    protected MenuList m_menuList;

    public EngineStatePauseMenu(Engine engine)
        : base(engine)
    {
        ...

        Point temp = m_engine.GraphicsDevice.Viewport.TitleSafeArea.Center;
        m_menuList = new MenuList(new List<string>(c_menuItems),
                                  new Vector2(temp.X, temp.Y - 100));
        m_menuList.BaseColor = Color.Brown;
        m_menuList.SelectedColor = Color.Red;
        m_menuList.Font = FontEnum.Kootenay48;
        m_menuList.ItemSpacing = 100;
        m_menuList.DrawDepth = Constants.DepthMainMenuText + 0.002f;
    }

    public override void update(Microsoft.Xna.Framework.GameTime gameTime)
    {
        ...

        if (InputSet.getInstance().getButton(InputsEnum.CONFIRM_BUTTON))
        {
            InputSet.getInstance().setAllToggles();

            switch (m_menuList.SelectedString)
            {
                case "Return to Game":
                    EngineManager.popState();
                    return;
                case "Settings":
                    // TODO
                    break;
                case "Quit Game":
                    EngineManager.popState();
                  ((EngineStateGameplay)EngineManager.peekAtState()).cleanup();
                    EngineManager.popState();
                    EngineManager.pushState(new EngineStateMainMenu(m_engine));
                    return;
            }
        }

        if (InputSet.getInstance().getLeftDirectionalY() < 0)
        {
            m_menuList.selectNextItem();
            InputSet.getInstance().setStick(InputsEnum.LEFT_DIRECTIONAL, 5);
        }

        if (InputSet.getInstance().getLeftDirectionalY() > 0)
        {
            m_menuList.selectPreviousItem();
            InputSet.getInstance().setStick(InputsEnum.LEFT_DIRECTIONAL, 5);
        }
    }

    public override void draw()
    {
        ...

        m_menuList.draw();
    }
}
```

i. Note the use of “sticks” as opposed to “toggles” when moving through the menu. A “stick” tells an input not to activate for a fixed number of frames. When that number is up, the input works normally.

Collision Detection

The engine uses a QuadTree-based collision detection system, so currently everything has a rectangular collider. The ICollider class is registered with a CollisionDetector object, and two types of queries can be made to the Collision Detector. The first is a direct query for a list of all colliders in a rectangular region; it is then up to the caller to decide what to do with this information. The second type of query is a request to handle movement. In this type of query, the caller provides the detector with a desired change in position and the detector attempts to determine how much of that change should be allowed. If it detects a collision over that range of movement, it may forward that information to the CollisionHandler, which looks up the two types of objects involved and determines what should happen. Usually what happens is a simple scaling back of velocity.

The CollisionHandler works well at reasonably low speeds for the characters, but the current algorithm in place will have problems if you have objects moving very quickly. For example, if an object is moving diagonally very quickly, the detector may think that its corner clipped something which it would not have. I would recommend simply trying it and if you have problems you can improve upon it.

The algorithm was intentionally designed to let characters “slide” along other objects, for instance, if you are holding Down+Right and the character’s bottom is touching a rectangle top, the character will move Right with the full Right velocity. However, this type of system produced some drawbacks, for instance if you try and charge into a convex corner, you will stop shy of the corner.

Tutorial: Content Pipeline and Content Editor

In this tutorial, we will explore how to add new data types to the game, and how to create instances of those data types using the provided editor. We will create a very basic inventory system.

1. Adding New Data Types
   a. We'd like to add basic inventory information to our game, so that our character's attack damage is based on his currently equipped weapon. Preferably we'd like to store information about items in XML so that we can attach them to different monsters, treasure tables, stores, etc. All data stored in XML should be a type in the CS8803AGAGameLibrary project, so we make some new types there:

```
public class Item
{
    public string name { get; set; }
    public int valueInGold { get; set; }
}

public class Weapon : Item
{
    public int damage { get; set; }
}

public class Armor : Item
{
    public int protection { get; set; }
}
```

   b. And we add these types as properties in our CharacterInfo class:

```
        [ContentSerializer(Optional = true)]
        public Weapon equippedWeapon { get; set; }

        [ContentSerializer(Optional=true)]
        public Item[] inventory { get; set; }
```

   The [ContentSerializer(Optional=true)] tags are extremely important here. If you don't include them and try to compile the game, all of the old CharacterInfo XML files will be considered broken by the compiler because they are missing a required tag.

   c. Now we'll hook up the weapon damage in code. In the CharacterController class:

```
public Weapon EquippedWeapon { get; protected set; }

...

public static CharacterController construct(CharacterInfo ci, Vector2 startpos,
bool playerControlled)
{

    ...

    cc.EquippedWeapon = ci.equippedWeapon;

    return cc;
}
```

d. And we modify actions/ActionAttack to poll the weapon for damage:

```
public override void execute(object sendingCharacterController)
{
    ...

    int damageAmt = cc.EquippedWeapon == null ? 1 : cc.EquippedWeapon.damage;
    GameplayManager.ActiveArea.add(new DamageTrigger(bounds, cc, damageAmt));
}
```

2. Modifying the Data with the Editor
    a. Now that we have code support, we next want to modify an existing character info (the player character's, which is in "CS8803AGA/Content/Characters/Jason.xml") to have a new weapon which does 2 damage (thus killing the NPCs in one hit instead of two).
    b. To run the editor, simply run the CS8803AGAEditor project. Then load the Jason.xml file.
    c. The first thing to notice is that the "equippedWeapon" property is grayed out, and can't be edited. This is because it is a class, and the default value for class types is null. Unfortunately, somebody at Microsoft did not have the foresight to recognize that we would want to instantiate classes in the PropertyGrid (that's the name of the key/value pairs component you're using), so you are almost stuck with editing it by hand. However, there is a patchwork hackish fix for this called "SmartExpandableConverter". This is a custom type converter which can "convert" a string value into an instance of the class, but really it ignores the string and uses reflection to attempt to use the default constructor to create an instance of the class. If the class has subclasses or is an interface, it provides a prompt window and allows you to select which subclass to instantiate (because this isn't how things are supposed to work, you actually have to select it thrice… if anyone finds a non-hackish fix for this, please let me know). Anyway, to use SmartExpandableConverter, we attach it to either the class or the individual property. I'll show the code for that shortly.
    d. When we attempt to edit the inventory property, we get an ellipsis indicating that we can use the ArrayEditor (or in other cases the CollectionEditor) for modifying the objects inside that property. Open that up and click Add to create a new Item instance. A window will appear which lets you select which type of Item you'd like to create – this is another non-Microsoft feature which I have set to apply automatically to all Array properties (if you would like it for List or other collection types you will have to apply it manually for each). Fill in some sample items with names and save.
    e. Once you close the ArrayEditor, you'll see that inventory now has a "+" symbol next to it, indicating it is expandable. However, if you expand it, you'll get a bunch of rows indicating not much, just the class type of the items. It would be nice if that looked better and also allowed editing of the individual items without reopening the

ArrayEditor. Both of these are easily doable.

```
// Tells the Editor that Item objects should be expandable in the grid
[TypeConverter(typeof(ExpandableObjectConverter))]
public class Item
{
    // Tells the Editor that when this property changes, update the
    //  property used to display the whole object ("name" gets this
    //  because it is used in the ToString method below, so when name
    //  changes the appearance of the whole object changes in the field)
    // This isn't vital, just nice, so you can ignore it if you want.
    [NotifyParentProperty(true)]
    public string name { get; set; }
    public int valueInGold { get; set; }

    // Tells the Editor to display this String instead of the ugly type
    public override string ToString()
    {
        return String.Format("{0}: {1}", this.GetType().Name, name);
    }
}

// Allows us to enter a dummy string to trigger the construction of a
//  Weapon object even when the property is null.
[TypeConverter(typeof(SmartExpandableConverter<Weapon>))]
public class Weapon : Item
{
    public int damage { get; set; }
}
```

f. Now boot up the editor and see what has changed. You can now edit the "equippedWeapon" field. Type in any string other than the empty string and hit enter, and it will instantiate a Weapon (this is courtesy of SmartExpandableConverter). To return it to null, erase everything in that field and hit enter. It should also be expandable now so that you can edit its properties without going to a separate editor window. Next, you can also expand the "inventory" property and instead of seeing ugly fully qualified type names, you see the class name followed by the item name. These are also individually expandable so their properties can be edited, but if you want to add or remove them you need to go back to the ArrayEditor.

g. Create an "equippedWeapon" which deals damage 2 and save. Boot up the game, you should be able to kill NPCs in one hit.

3. Two Comments

   a. You should try to create a new type from scratch and load it in the game code. As per the opening description of the Content Pipeline, you'll then have to drop that XML somewhere into the Content folder. It contains type information in it, so to retrieve it in game code you call Content.Load<T>(assetPathExcludingExtension). You can also do it via the GlobalHelper.loadContent<T>() function. Note that the ContentManager caches these objects, so I believe that you shouldn't edit them if there's a chance you will load them more than once.

   b. Truthfully, storing all of the Item and Weapon information inside the CharacterInfo is probably not the best way to do things. You probably want to create standalone Item XML files and then provide their assetPaths within files that use them, so CharacterInfo's "equippedWeapon" would actually just be a string. You would then load that asset via the string at runtime. This allows you to reuse items across different data assets. However, I felt this example was a good way to illustrate the editor, subclasses, expandability, etc.

Animation Sets

The animations in the game are stored in a class called a GameLibrary/animation/AnimationSet. An AnimationSet is simply a container for Animations. Each animation is a set of sprites from a sprite sheet which play in a certain order. The Animation contains some metadata about how interruptible it is, whether it loops, what should happen when it is finished, etc., while each Sprite object stores information about its location on the sprite sheet, how many frames it should display for, events associated with it, etc. These classes are important so you should take a look at them.

Currently, the only way to use an AnimationSet is via an AnimationController. Constructing an AnimationController requires two arguments: a path to the XML for the AnimationSet, and a path to the image file which the AnimationSet uses. These two are disassociated if, for instance, you wanted to have two different-colored goblins which had all of the exact same metadata but two separate sprite sheets.

Sprites can contain ActionInfos, which can contain events to fire when a particular sprite in an animation is reached. Such logic could be game logic, visual effect, sound, etc. For instance, left and right attacks of the player character have an ActionInfoAttack which places a DamageTrigger in the world to hurt other characters. Please note that in industry, it is generally frowned upon to have animations control game logic, so if you prefer to do that you should remove these attack actions and avoid putting game logic into these. In my opinion, I think it is acceptable for simple games because it provides an easier (though more constrained) way of syncing game logic with animations.

A note about pixel bleed: you may notice that the player character has some nasty shadow/side effects going on when he moves around. That's because of pixel bleed. When scaling images up, XNA uses linear interpolation over nearby pixels. Unfortunately, for sprites on a sprite sheet, it means that pixels outside of the source rectangle get included in the interpolation – thus the weird effects. There are a few ways around this:

1. Turn off linear interpolation. But then scaled-up images look like crap. Don't do this.
2. Scale the actual image file in paint or photoshop or whatever using whatever kind of interpolation you want, make sure it's clean, then don't scale the image at runtime.
3. Add gutter pixels to the sprite sheet, in other words, make sure that there is at least an empty boundary region of at least 1 pixel between sprites. Tedious if you already have the spritesheet made, but otherwise a relatively easy solution.

Currently, the CharacterController and PlayerController classes expect certain Animations to be present, e.g. "left," "attackright", etc. You'll have to modify them to get appropriate behavior for your game.

Tutorial: Audio and Animation Actions

This tutorial will explain how sound is played through the engine and how Actions can be attached to animations in the editor. A sound effect will be added to the player character's attack.

1. Creating an Action to play sounds
    a. We want to be able to support the playing of sounds attached to locations in our animations for things like sword slashes or footfalls. To do this, we will use an Action which plays sounds, and we will attach it to sprites in our animation metadata.
    b. Add a new type of ActionInfo to GameLibrary/actions. Call it "ActionInfoPlaySound.cs":

```
using System;
using Microsoft.Xna.Framework;

namespace CS8803AGAGameLibrary.actions
{
    public class ActionInfoPlaySound : AActionInfo
    {
        /// <summary>
        /// Cue in XACT project to play.
        /// </summary>
        public string cue { get; set; }
    }
}
```

    c. Next, we need code support for the PlaySound type. First, make a CS8803AGA/actions/ActionPlaySound class which knows how to play a sound:

```
using CS8803AGA.audio;
using CS8803AGAGameLibrary.actions;

namespace CS8803AGA.actions
{
    /// <summary>
    /// Action which plays a sound.
    /// </summary>
    public class ActionPlaySound : AAction<ActionInfoPlaySound>
    {
        public ActionPlaySound(ActionInfoPlaySound info) : base(info)
        {
            // nch
        }

        public override void execute(object source)
        {
            // source contains the Controller whose AnimationController
            // created the action, which we don't need to play a sound

            SoundEngine.getInstance().playCue(m_info.cue);
        }
    }
}
```

    d. Finally, modify CS8803AGA/actions/ActionFactory to support this new type of action.
2. Adding the ActionInfo to the data
    a. Add a PlaySound ActionInfo to Jason's animation set. This can be done either by hand-editing the XML or running the CS8803AGAEditor project. Using the Editor is the preferred method to do this. Edit "CS8803AGA/Content/Animation/Data/Jason.xml," the "attackleft" animation, the 3rd sprite (index 2), and modify the Action property to

include a new ActionInfo with type "PlaySound" and cue "bang_3", which is a cue set up in our XACT sound project. Make sure to save.

b. Alternatively, we can modify the file by hand. Open "CS8803AGA/Content/Animation/Data/Jason.xml" and make the addition in red:

```
<Item>
  <name>attackleft</name>
  <sprites>
    <Item>
      <box>0 128 32 32</box>
      <duration>7</duration>
      <action Null="true" />
      <center>16 144</center>
      <offset>0 0</offset>
    </Item>
    <Item>
      <box>32 128 32 32</box>
      <duration>7</duration>
      <action Null="true" />
      <center>48 144</center>
      <offset>0 0</offset>
    </Item>
    <Item>
      <box>64 128 41 32</box>
      <duration>7</duration>
      <action>
        <Item Type="actions:ActionInfoAttack">
          <location>-20 -10 17 23</location>
        </Item>
        <Item Type="actions:ActionInfoPlaySound">
          <cue>bang_3</cue>
        </Item>
      </action>
      <center>84 144</center>
      <offset>-8 0</offset>
    </Item>
```

3. Try it out – boot up the game and swing to the left. You should hear a sound.

In order to add new sounds to the game, you'll have to add them as cues to the XACT (Cross-Platform Audio Creation Tool) project. This tool is usually found in Start->Programs->Microsoft XNA Game Studio 3.1->Tools. A guide to using XACT can be found here: http://msdn.microsoft.com/en-us/library/ff827590.aspx. The project file can be found in "CS8803AGA/Content/Audio/sound.xap"

If you're interested in playing MIDI, here is a guide: http//social.msdn.microsoft.com/Forums/en-US/csharpgeneral/thread/161497fa-40c6-4a7b-92d1-b782aa92ea71

Miscellaneous Notes

- For simplicity of development, the engine is currently fixed at 30 fps. Standard XNA convention is to use real-time calculations and update/draw as fast as the hardware can handle it. If you want to do this, you'll be making changes in all of the animation classes, movement, damage… pretty much everywhere.
- If you want 3D graphics, I would completely remove all rendering code and convert the platform back into a single update-then-draw thread. It might be possible to just delete the rendering folder, see where you get compile errors, and go from there. If you want to start from scratch, you could drop in some of the things which will work in 3D – EngineManager, devices folder, utilities folder.
- If you want scrolling areas, you should set the Camera into the GlobalHelper and update its position each frame. If your areas are very large, I would recommend you create a separate QuadTree which stores graphical coordinates (as opposed to collision coordinates) and query it for the size of the screen each frame so that you aren't sending a bunch of draw commands which are going to get culled by XNA anyway.
- If you try it to run on Xbox (requires a Creator's Club Membership, but as a student there are places you can get one for free) but get poor performance, replace "foreach" loops and anonymous functions everywhere you can, load all content that you need up front, and try to limit allocating heap memory during main game loops. Manually perform a garbage collect whenever you make state transitions or can stand a moment of lag. The reason for all of this is that the Xbox's garbage collector sucks and will slow down your game if not treated with care. If you hand-load XML data during runtime (shouldn't need to with the 3.1 Content Pipeline, but if you really want to you can), you should use the ManagedXml class inside of a "using" block.
- If you would prefer an event-driven system for input handling, this shouldn't be too hard to do. I would recommend adding an interface for input events to IEngineState so that the handling is encapsulated uniquely for each engine state.
- If you want to save settings or options for your game, you might take a look at the Settings class.
- A platformer wouldn't be too difficult, just kill the area and world code, implement a scrolling camera, and add gravity.
- If you want a 3rd dimension for jumping, projectiles, cliffs, etc. (e.g. Castle Crashers), you could probably keep using the 2D collision detector and add a MinZ and MaxZ to all colliders. Then get all collisions on the plane and filter them for collisions in Z. Draw depth would still be based on the lower edge of the collision box on the ground. Alternatively, you could go full-blown 3D and switch it to an OctTree or use another data structure altogether.
- If you'd like collision detection on arbitrary convex 2D polygons, let Prof. Riedl know and I can give out some (reasonably old and ugly but highly effective) code for it. I'd still recommend using the QuadTree as a high-level collision detection (to narrow down pairs of objects which are needed for testing) because this code can be slow if used over all pairs of objects. For those interested, the code is for a separating-axis detection algorithm.
- You may find utilities/RandomManager useful. It can generate numbers from a normal distribution rather than just a uniform distribution.

Have fun!