Contents

Introduction to Game Development with Xamarin

10/3/2018 • 11 minutes to read • Edit Online

Developing games can be very exciting, especially given how easy it can be to publish your work on mobile platforms. This article discusses concepts and technologies related to game development that will help you create games, whether your goal is to create a high-quality AAA game or just to program for fun.

This article covers the following topics:

- **Game vs. non-game programming concepts** – We'll explore some concepts that are either unique to game development, or are shared with other types of development but deserve emphasis here due to their importance.
- **Game development team** – This section looks at the various roles in a team of game developers.
- **Creating a game idea** – This section can help you create a new game idea – the first step in making a new game.
- **Game development technology** – Here we'll list some of the cross-platform technologies available that can improve your productivity as a game developer.

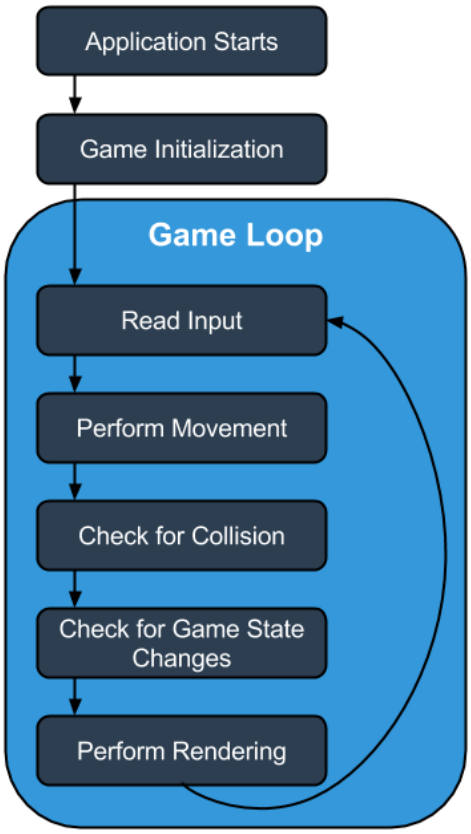
Game vs. Non-Game Programming Concepts

Programmers moving into game development are often confronted with new concepts and development patterns. This section presents a high-level view of some of these concepts.

The Game Loop

A typical game requires constant movement or change to be happening on the screen in response to both user interaction and automatic game logic. This is achieved through what is typically referred to as a *game loop*. A game loop is some type of looping statement (such as a while-loop) which runs at a very high frequency, such as 30 or 60 *frames per second*.

The following is a diagram of a simple game loop:



The technologies we discuss below will abstract away the actual while-loop, but despite this abstraction the concept of every-frame updates will be present.

Code performance can take priority in even the simplest of games. For example: a function which takes 10 milliseconds to execute could have a significant impact on the performance of a game – especially if it's called more than once per frame. If your game is running at 30 frames per second then that means each frame must execute in under 33 milliseconds. By contrast, such a function may not even be noticeable if it only executes in response to a button click in a non-game application.

Common types of logic that may be performed every-frame include:

- **Reading Input** – The game may need to check if the user has interacted with the game by checking input hardware, such as the touch screen, keyboard, mouse, or game controller.
- **Movement** – Objects which move from one place to another will typically move a very small amount every frame to give the illusion of fluid motion.
- **Collision** – Many games require the frequent testing of whether various objects are overlapping or intersecting. We'll cover collision in more depth in a later section in this article. Movement and collision may be handled by a dedicated physics simulation system.
- **Checking for game-specific conditions** – The state of a game may be controlled by certain conditions, such as whether the player has earned enough points or whether allotted time has run out.
- **AI behavior** – Every-frame logic that may be used to control the behavior of objects which are not controlled by the player, such as the patrolling of an enemy or the movement of opponent drivers around a racetrack.
- **Rendering** – Most games will update what is displayed on screen every frame. This may be done in response to changes which have impact on game play (such as a character moving through a level) or simply to provide visual polish (such as falling snow or animated icons).

Keep in mind that many of the activities listed above can change the state of the entire application, whereas many non-game apps tend to change state in response to events being raised.

Content Loading and Unloading

Manually loading and unloading (or disposing) content may be needed depending on which technology you are using in development. Manually loading and unloading of assets may be necessary for a number of reasons:

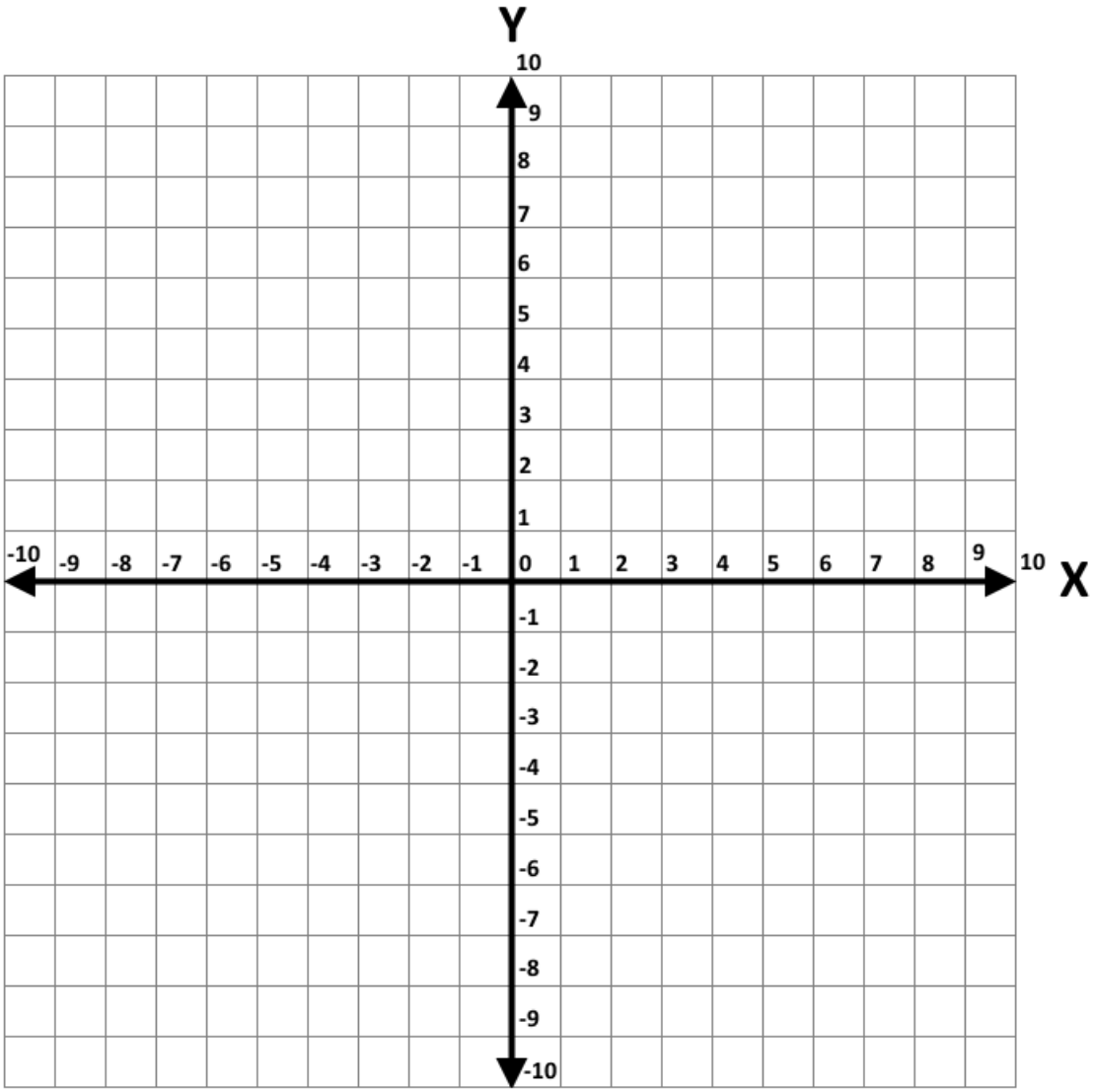
- Assets may take a long time to load relative to the length of a single frame. Some assets may even take seconds to load, which would severely disrupt the experience if loaded mid gameplay. If the load time is especially lengthy (such as more than a second or two) you may want to show an animated loading screen or progress bar.
- Assets can consume a lot of RAM, requiring active management of what is loaded to fit within what is provided by the game's target platforms.
- Games may need to display more assets than can fit in RAM. "Open World" games often include large environments which players can navigate through seamlessly – that is with no loading screens. In this case you may need to create a custom system for streaming content in and managing memory usage.

Custom file formats may need processing at load time, requiring custom loading code.

Math

Many games require more advanced mathematics than non-game applications. Of course, the level of math depends on the complexity of the game. In general 3D games require more math than 2D. Fortunately you can always get started with simple games and learn as you go. Game development can be a great way to learn math!

If you're familiar with the Cartesian plane – that is using X and Y coordinates to position objects – then you know enough to get started with game development. The following shows a Cartesian plane with positive Y pointing upward:



IMPORTANT

Some engines/APIs use a coordinate system where increasing an object's Y value will move it down, while other systems use a coordinate system where positive Y is up. Keep this in mind if you are moving between systems. Trigonometric functions (such as Sine and Cosine) are commonly used in 2D games which implement any form of rotation.

If you are planning on making a 3D game then you will likely need to be familiar with concepts from Linear Algebra (for rotation and movement in 3D space) as well as some Calculus (for implementing acceleration).

Content Pipelines

The term *content pipeline* refers to the process that a file takes to get from its format when authored (such as a .png image file) to its final format when used in a game. The ending format depends on which type of content is being used as well as which technology is being used to present the content.

Some content pipelines may be very fast and require no manual effort. For example, most game engines and APIs can load the .png file format in its unprocessed format. On the other hand, more complicated formats (such as 3D models) may need to be processed into a different format before being loaded, and this processing can take some time depending on the asset's size and complexity.

Game Development Teams

Game development introduces new roles and titles for individuals involved in the process. Most game developers are not able to satisfy the broad set of skills required to release a full game, so a number of disciplines exist. Keep in mind that this is not a full list of areas of development – just some of the more common ones.

- **Programmer** – Most people reading this article will fall into this category. The role of a programmer in game development is similar to a programmer's role in a non-game application. Responsibilities include writing logic to control the flow of a game, developing systems for common tasks in the context of a given project, adding and displaying content, and – of course – fixing bugs.
- **2D artist** – 2D artists are responsible for creating *2D assets*. These include image files for the game's GUI, particles, environments, and characters. If the game you are developing is 3D, then 2D artists may not be responsible for environments and characters. You can find free art for your game at http://opengameart.org/ .
- **3D artists** – 3D artists are responsible for creating *3D assets*. These include 3D models for environments, characters, and props (furniture, plants, and other inanimate objects). Some teams differentiate between 3D artists and 3D animators depending on the size of the team. You can find free 3D art for your game at http://opengameart.org/ .
- **Game Designer** – Game designers are responsible for defining how the game is played. This can include high-level decisions such as the setting of the game, the overall goal of the game, and how a player progresses through the game. Game designers can also be involved in very detailed decisions such as mapping input to actions, defining coefficients for movement or level-ups, and designing level layout. Keep in mind that the term *designer* may refer to a game designer or a visual designer depending on the context.
- **Sound Designer** – Sound designers are responsible for a game's audio assets. Some teams may differentiate between individuals responsible for creating sound effects and composers, while smaller teams may have a single individual responsible for all audio.

Creating a Game Idea

Designing a game may appear to be easy to do – after all the only requirement is "make something fun." Unfortunately, many developers find themselves at a loss when it comes time to create an idea from which to launch development.

The discipline of game design is not easily explained, and requires practice to improve just as the art or programming does, but this section can help you start down the path.

New developers should start small. It can be difficult to resist the temptation to remake a large, modern video game, but smaller games can be a better learning environment and the faster progress makes for a more rewarding experience.

Many games, both for the purpose of learning as well as commercial games, begin as an improvement or modification to an existing game. One way to generate ideas is to look at other games for inspiration. For example, you can consider a game that you personally like and try to identify what characteristics about the game play make it fun. It may be exploration, mastery of the game's mechanics, or progressing through a story. Don't forget to consider "retro" games as well when searching for new ideas.

Another technique for generating new ideas is to consider a specific genre, such as puzzle games, strategy games, or platformers. A genre familiar to the developer might provide a good starting point.

Remaking existing games is also an educational experience, although this may limit the finished product's commercial viability. The process of creating a game, even one which is an accurate clone, provides a valuable educational experience.

Game Development Technology

Developers using Xamarin.Android and Xamarin.iOS have a wide range of technologies available to them to assist in game development. This section will discuss some of the most popular cross-platform solutions.

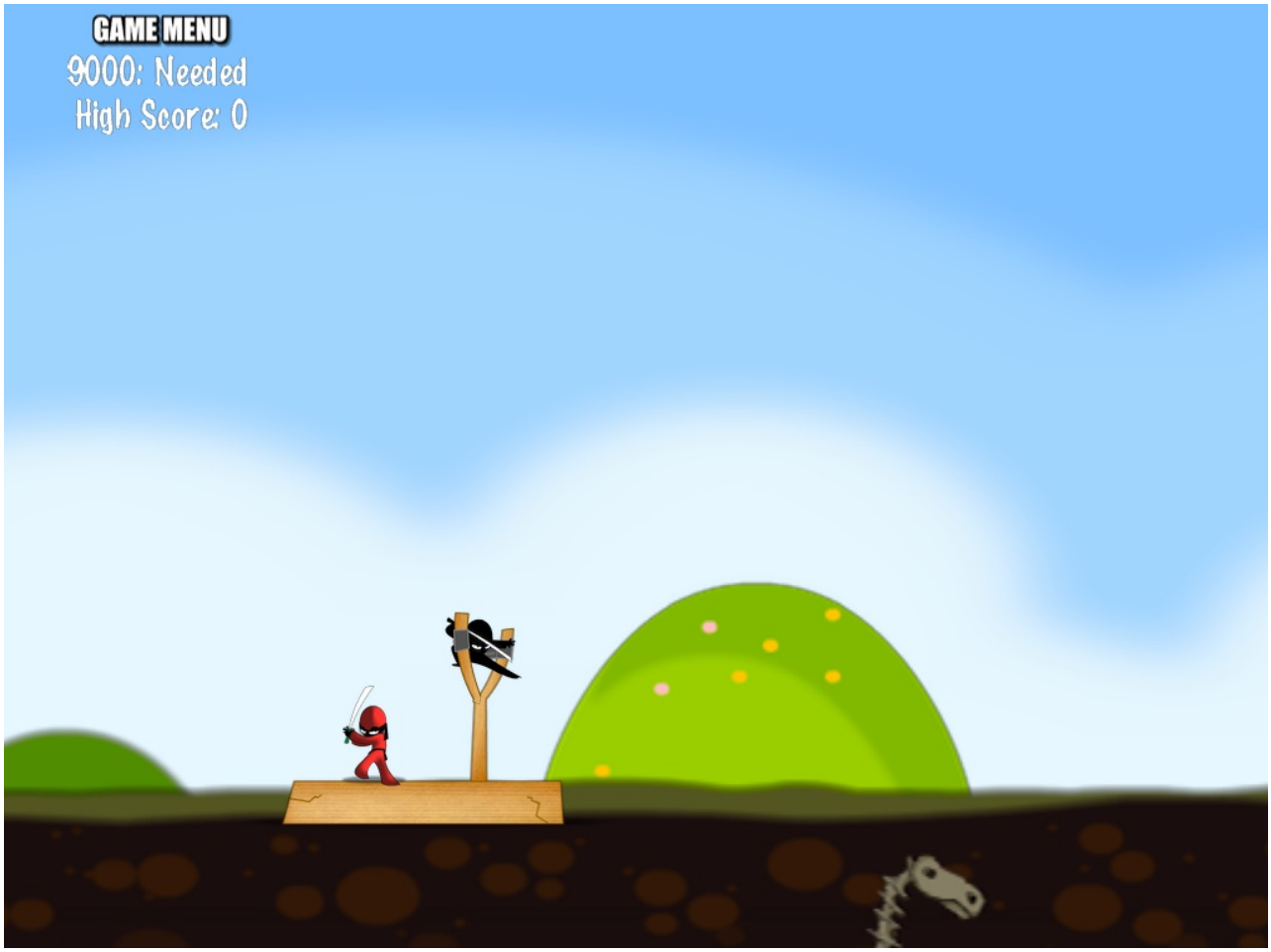
CocosSharp

CocosSharp is an open-source, cross-platform version of the Cocos 2D game engine. The engine provides access to Android, iOS, Mac OS X, Windows Desktop, Windows RT and Windows Phone.

CocosSharp focuses on a simple programmer API for 2D game development. The growth in gaming on mobile devices has helped to reignite the popularity of 2D game development making CocosSharp viable technology for hobby and commercial projects alike. It is provided as source code or .dll files (which can be obtained through NuGet) but it does not offer a visual editor; therefore, any interaction with the CocosSharp engine requires knowledge of programming.

To get started with CocosSharp, check out our CocosSharp guides.

The game Angry Ninjas is created with CocosSharp, and it can be a good starting point if you're looking for an already-running game for multiple platforms:



You can download it and get more information at the AngryNinjas Github page.

MonoGame

MonoGame is an open-source, cross platform version of Microsoft's XNA API. MonoGame can be used to make games for iOS, Android, Mac OS X, Linux, Windows, Windows RT, PS4, PSVita, Xbox One, and Switch.

Unlike CocosSharp, MonoGame is technically not a game engine, but rather a game development API. This means that working with MonoGame requires directly managing game objects, manually drawing objects, and implementing common objects such as cameras and *scene graphs* (the parent child hierarchy between game objects). To help understand the distinction, consider that CocosSharp is built on top of MonoGame. MonoGame generalizes some of the platform-specific technology, such as graphics, rendering, and audio, while CocosSharp provides code for organizing and implementing game logic.

MonoGame does not offer a standard visual development environment, so working with MonoGame requires programming knowledge.

Notable examples of games using MonoGame include:

FEZ:

Bastion:

To start working with MonoGame, head over to our MonoGame Guides.

UrhoSharp

UrhoSharp is a cross-platform high-level 3D and 2D engine that can be used to create animated 3D and 2D scenes

for your applications using geometries, materials, lights and cameras.

Check out the UrhoSharp Guides to get started.

Additional Technology

The technologies highlighted above is only a sample of the technologies available. Other notable technologies include:

- **Sprite Kit** – Xamarin provides support for Apple's Sprite Kit game framework, which gives you access to all of the functionality of the native API. Since Sprite Kit is technology created by Apple, it provides deep integration with the rest of the iOS ecosystem. Of course, Sprite Kit is not cross-platform so it cannot be used on Android. For more information on using Sprite Kit, see this post: http://blog.xamarin.com/make-games-with-xamarin.ios-and-sprite-kit/
- **Scene Kit** – Xamarin also provides support for Apple's Scene Kit framework, which simplifies implementing 3D graphics into iOS apps. Scene Kit is also technology provided by Apple, so it has both the integration and platform-specific considerations mentioned above for Sprite Kit. For more information on Scene Kit, see this post: http://blog.xamarin.com/3d-in-ios-8-with-scene-kit/
- **OpenTK –** OpenTK (which stands for Open Tool Kit) provides low-level OpenGL access to iOS, Apple, and Mac hardware. For more information on OpenTK, see the main page at: http://www.opentk.com/

Summary

This article covers the main concepts of game development and provides information on how to get started making your first game. Once you've finished this article, the next steps are to pick your technology and begin working through our series of tutorials linked in the appropriate sections above.

Related Links

- CocosSharp Guides
- MonoGame Guides

- UrhoSharp Guides

SkiaSharp Graphics in Xamarin.Forms

12/7/2018 • 2 minutes to read • Edit Online

Download the sample

Use SkiaSharp for 2D graphics in your Xamarin.Forms applications

SkiaSharp is a 2D graphics system for .NET and C# powered by the open-source Skia graphics engine that is used extensively in Google products. You can use SkiaSharp in your Xamarin.Forms applications to draw 2D vector graphics, bitmaps, and text. See the 2D Drawing guide for more general information about the SkiaSharp library and some other tutorials.

This guide assumes that you are familiar with Xamarin.Forms programming.

Webinar: SkiaSharp for Xamarin.Forms

SkiaSharp Preliminaries

SkiaSharp for Xamarin.Forms is packaged as a NuGet package. After you've created a Xamarin.Forms solution in Visual Studio or Visual Studio for Mac, you can use the NuGet package manager to search for the **SkiaSharp.Views.Forms** package and add it to your solution. If you check the **References** section of each project after adding SkiaSharp, you can see that various **SkiaSharp** libraries have been added to each of the projects in the solution.

If your Xamarin.Forms application targets iOS, use the project properties page to change the minimum deployment target to iOS 8.0.

In any C# page that uses SkiaSharp you'll want to include a `using` directive for the `SkiaSharp` namespace, which encompasses all the SkiaSharp classes, structures, and enumerations that you'll use in your graphics programming. You'll also want a `using` directive for the `SkiaSharp.Views.Forms` namespace for the classes specific to Xamarin.Forms. This is a much smaller namespace, with the most important class being `SKCanvasView`. This class derives from the Xamarin.Forms `View` class and hosts your SkiaSharp graphics output.

> **IMPORTANT**
> The `SkiaSharp.Views.Forms` namespace also contains an `SKGLView` class that derives from `View` but uses OpenGL for rendering graphics. For purposes of simplicity, this guide restricts itself to `SKCanvasView`, but using `SKGLView` instead is quite similar.

SkiaSharp Drawing Basics

Some of the simplest graphics figures you can draw with SkiaSharp are circles, ovals, and rectangles. In displaying these figures, you will learn about SkiaSharp coordinates, sizes, and colors. The display of text and bitmaps is more complex, but these articles also introduce those techniques.

SkiaSharp Lines and Paths

A graphics path is a series of connected straight lines and curves. Paths can be stroked, filled, or both. This article encompasses many aspects of line drawing, including stroke ends and joins, and dashed and dotted lines, but stops

short of curve geometries.

SkiaSharp Transforms

Transforms allow graphics objects to be uniformly translated, scaled, rotated, or skewed. This article also shows how you can use a standard 3-by-3 transform matrix for creating non-affine transforms and applying transforms to paths.

SkiaSharp Curves and Paths

The exploration of paths continues with adding curves to a path objects, and exploiting other powerful path features. You'll see how you can specify an entire path in a concise text string, how to use path effects, and how to dig into path internals.

SkiaSharp Bitmaps

Bitmaps are rectangular arrays of bits corresponding to the pixels of a display device. This series of articles shows how to load, save, display, create, draw on, animate, and access the bits of SkiaSharp bitmaps.

SkiaSharp Effects

Effects are properties that alter the normal display of graphics, including linear and circular gradients, bitmap tiling, blend modes, blur, and others.

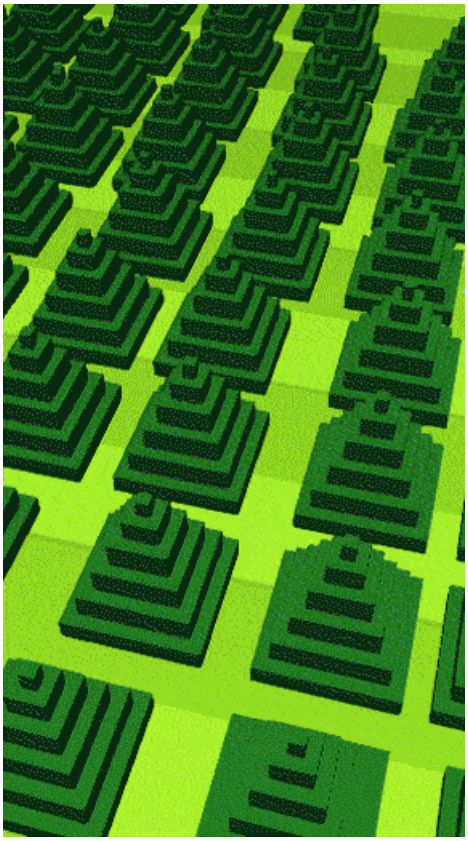
Related Links

- SkiaSharp APIs
- SkiaSharpFormsDemos (sample)
- SkiaSharp with Xamarin.Forms Webinar (video)

UrhoSharp - 3D/2D Engine

10/10/2018 • 2 minutes to read • Edit Online

UrhoSharp is a cross-platform high-level 3D and 2D engine that can be used to create animated 3D and 2D scenes for your applications using geometries, materials, lights and cameras.



UrhoSharp is distributed as a NuGet package that can be installed on either Visual Studio or Visual Studio for Mac and can be used to target any of the following platforms: Android, MacOS, iOS, tvOS and Windows.

An Introduction to UrhoSharp

This article provides a high-level overview of UrhoSharp and its capabilities for 3D visualization and for use in simple 3D games.

Using UrhoSharp

In this document we describe the core concepts of UrhoSharp that you would use to build a game or create a 3D visualization for your application.

Urho and Your Platform

These guides describe the setup instructions for Urho on each target platform and describe ways to integrate Urho with your existing Android and iOS applications.

Programming UrhoSharp With F#

This guide walks through the creation of a simple "Hello, World!" UrhoSharp solution using F# and Visual Studio for Mac.

API Documentation

You can browse the API documentation for UrhoSharp on our web site.

Samples

We have created samples on GitHub illustrating how to use UrhoSharp.

- **FeatureSamples** shows more than 40 individual samples that showcase specific features of Urho.
- **SamplyGame** is a sample implementation of the Shooty Skies game.
- **FormsSample** showcases how to use UrhoSharp in Xamarin.Forms applications.

All the samples run on Android, iOS, Mac and Windows.

Copyright

License

An Introduction to UrhoSharp

10/10/2018 • 6 minutes to read • Edit Online

UrhoSharp is a powerful 3D Game Engine for Xamarin and .NET developers. It is similar in spirit to Apple's SceneKit and SpriteKit and include physics, navigation, networking and much more while still being cross platform.

It is a .NET binding to the Urho3D engine and allows developers to write cross platform code that can target Android, iOS, Windows and Mac with the same codebase and can render to both OpenGL and Direct3D systems.

UrhoSharp is a game engine with a lot of functionality out of the box:

- Powerful 3D graphic rendering
- Physics simulation (using the Bullet library)
- Scene handling
- Await/Async support
- Friendly Actions API
- 2D integration into 3D scenes
- Font rendering with FreeType
- Client and server networking capabilities
- Import a wide range of assets (with Open Assets Library)
- Navigation mesh and pathfinding (using Recast/Detour)
- Convex hull generation for collision detection (using StanHull)
- Audio playback (with **libvorbis**)

Getting Started

UrhoSharp is conveniently distributed as a NuGet package and it can be added to your C# or F# projects that target Windows, Mac, Android or iOS. The NuGet comes with both the libraries required to run your program, as well as the basic assets (CoreData) used by the engine.

Urho as a Portable Class Library

The Urho package can be consumed either from a platform-specific project, or from a Portable Class Library project, allowing you to reuse all of your code across all platforms. This means that all you would have to do on each platform is to write your platform specific entry point, and then transfer control to your shared game code.

Samples

You can get a taste for the capabilities of Urho by opening in either Visual Studio for Mac or Visual Studio the Sample solution from:

https://github.com/xamarin/urho-samples

The default solution contains projects for Android, iOS, Windows and Mac. We have structured that solution so that we have a tiny platform specific launcher, and all of the sample code and game code lives in a portable class library, illustrating how to maximize code reuse across all platforms.

Consult the Urho and Your Platform page for more information on how to create your own solutions.

Since all of the samples share a common set of user interface elements, the samples have abstracted the basic setup in this file:

https://github.com/xamarin/urho-samples/blob/master/FeatureSamples/Core/Sample.cs

This provides a Sample base class that handles some basic keystrokes and touch events, setups a camera, provides basic user interface elements, and this allows each sample to focus on the specific functionality that is being showcased.

The following sample shows what the engine is capable of doing:

- Samply Game a simple clone of ShootySkies.

While the other samples show individual properties of each sample.

Basic Structure

Your game should subclass the `Application` class, this is where you will setup your game (on the `Setup` method) and start your game (in the `Start` method). Then you construct your main user interface. We are going to walk through a small sample that shows the APIs to setup a 3D scene, some UI elements and attaching a simple behavior to it.

```
class MySample : Application {
    protected override void Start ()
    {
        CreateScene ();
        Input.KeyDown += (args) => {
            if (args.Key == Key.Esc) Exit ();
        };
    }

    async void CreateScene()
    {
        // UI text
        var helloText = new Text()
        {
            Value = "Hello World from MySample",
            HorizontalAlignment = HorizontalAlignment.Center,
            VerticalAlignment = VerticalAlignment.Center
        };
        helloText.SetColor(new Color(0f, 1f, 1f));
        helloText.SetFont(
            font: ResourceCache.GetFont("Fonts/Font.ttf"),
            size: 30);
        UI.Root.AddChild(helloText);

        // Create a top-level scene, must add the Octree
        // to visualize any 3D content.
        var scene = new Scene();
        scene.CreateComponent<Octree>();
        // Box
        Node boxNode = scene.CreateChild();
        boxNode.Position = new Vector3(0, 0, 5);
        boxNode.Rotation = new Quaternion(60, 0, 30);
        boxNode.SetScale(0f);
        StaticModel modelObject = boxNode.CreateComponent<StaticModel>();
        modelObject.Model = ResourceCache.GetModel("Models/Box.mdl");
        // Light
        Node lightNode = scene.CreateChild(name: "light");
        lightNode.SetDirection(new Vector3(0.6f, -1.0f, 0.8f));
        lightNode.CreateComponent<Light>();
        // Camera
        Node cameraNode = scene.CreateChild(name: "camera");
        Camera camera = cameraNode.CreateComponent<Camera>();
        // Viewport
        Renderer.SetViewport(0, new Viewport(scene, camera, null));
        // Perform some actions
        await boxNode.RunActionsAsync(
            new EaseBounceOut(new ScaleTo(duration: 1f, scale: 1)));
        await boxNode.RunActionsAsync(
            new RepeatForever(new RotateBy(duration: 1,
                deltaAngleX: 90, deltaAngleY: 0, deltaAngleZ: 0)));
     }
}
```

If you run this application, you will quickly discover that the runtime is complaining about your assets are not there. What you need to do is create a hierarchy in your project that starts with the special directory name "Data", and inside this, you would place the assets that you reference in your program. You must then set in the item properties for each asset the “Copy to Output Directory” to “Copy if Newer”, that will ensure that your data is there.

Let us explain what is going on here.

To launch your application you call the engine initialization function, followed by creating a new instance of your Application class, like this:

```
new MySample().Run();
```

The runtime will invoke the `Setup` and `Start` methods for you. If you override `Setup` you can configure the engine parameters (not show in this sample).

You must override `Start` as this will launch your game. In this method you will load your assets, connect event handlers, setup your scene and start any actions that you desire. In our sample, we both create a little bit of UI to show to the user as well as setting up a 3D scene.

The following piece of code uses the UI framework to create a text element and add it to your application:

```
// UI text
var helloText = new Text()
{
    Value = "Hello World from UrhoSharp",
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
helloText.SetColor(new Color(0f, 1f, 1f));
helloText.SetFont(
    font: ResourceCache.GetFont("Fonts/Font.ttf"),
    size: 30);
UI.Root.AddChild(helloText);
```

The UI framework is there to provide a very simple in-game user interface, and it works by adding new nodes to the `UI.Root` node.

The second part of our sample setups the main scene. This involves a number of steps, creating a 3D Scene, creating a 3D box in the screen, adding a light, a camera and a viewport. These are explored in more detail in the section Scene, Nodes, Components and Cameras.

The third part of our sample triggers a couple of actions. Actions are recipes that describe a particular effect, and once created they can be executed by a node on demand by calling the `RunActionAsync` method on a `Node`.

The first action scales the box with a bouncing effect and the second one rotates the box forever:

```
await boxNode.RunActionsAsync(
    new EaseBounceOut(new ScaleTo(duration: 1f, scale: 1)));
```

The above shows how the first action that we create is a `ScaleTo` action, this is merely a recipe that indicates that you want to scale for a second towards the value one the scale property of a node. This action in turn is wrapped around an easing action, the `EaseBounceOut` action. The easing actions distort the linear execution of an action and apply an effect, in this case it provides the bouncing-out effect. So our recipe could be written as:

```
var recipe = new EaseBounceOut(new ScaleTo(duration: 1f, scale: 1));
```

Once the recipe has been created, we execute the recipe:

```
await boxNode.RunActionsAsync (recipe)
```

The await indicates that the will want to resume execution after this line when the action completes. Once the action completes we trigger the second animation.

The Using UrhoSharp document explores in more depth the concepts behind Urho and how to structure your code to build a game.

Copyrights

Using UrhoSharp To Build A 3D Game

10/10/2018 • 21 minutes to read • Edit Online

Before you write your first game, you want to get familiarized with the basics: how to setup your scene, how to load resources (this contains your artwork) and how to create simple interactions for your game.

Scenes, Nodes, Components and Cameras

The scene model can be described as a component-based scene graph. The Scene consists of a hierarchy of scene nodes, starting from the root node, which also represents the whole scene. Each `Node` has a 3D transform (position, rotation and scale), a name, an ID, plus an arbitrary number of components. Components bring a node to life, they can make add a visual representation (`StaticModel`), they can emit sound (`SoundSource`), they can provide a collision boundary and so on.

You can create your scenes and setup nodes using the Urho Editor, or you can do things from your C# code. In this document we will explore setting things up using code, as they illustrate the elements necessary to get things to show up on your screen

In addition to setting up your scene, you need to setup a `Camera` , this is what determines what will get shown to the user.

Setting up your Scene

You would typically create this form your Start method:

```
var scene = new Scene ();
// Create the Octree component to the scene. This is required before
// adding any drawable components, or else nothing will show up. The
// default octree volume will be from -1000, -1000, -1000) to
//(1000, 1000, 1000) in world coordinates; it is also legal to place
// objects outside the volume but their visibility can then not be
// checked in a hierarchically optimizing manner
scene.CreateComponent<Octree> ();
// Create a child scene node (at world origin) and a StaticModel
// component into it. Set the StaticModel to show a simple plane mesh
// with a "stone" material. Note that naming the scene nodes is
// optional. Scale the scene node larger (100 x 100 world units)
var planeNode = scene.CreateChild("Plane");
planeNode.Scale = new Vector3 (100, 1, 100);
var planeObject = planeNode.CreateComponent<StaticModel> ();
planeObject.Model = ResourceCache.GetModel ("Models/Plane.mdl");
planeObject.SetMaterial(ResourceCache.GetMaterial("Materials/StoneTiled.xml"));
```

Components

Rendering 3D objects, sound playback, physics and scripted logic updates are all enabled by creating different Components into the nodes by calling `CreateComponent<T>()` . For example, setup your node and light component like this:

```
// Create a directional light to the world so that we can see something. The
// light scene node's orientation controls the light direction; we will use
// the SetDirection() function which calculates the orientation from a forward
// direction vector.
// The light will use default settings (white light, no shadows)
var lightNode = scene.CreateChild("DirectionalLight");
lightNode.SetDirection (new Vector3(0.6f, -1.0f, 0.8f));
```

We have created above a node with the name "`DirectionalLight`" and set a direction for it, but nothing else. Now, we can turn the above node into a light-emitting node, by attaching a `Light` component to it, with `CreateComponent`:

```
var light = lightNode.CreateComponent<Light>();
```

Components created into the `Scene` itself have a special role: to implement scene-wide functionality. They should be created before all other components, and include the following:

- `Octree`: implements spatial partitioning and accelerated visibility queries. Without this 3D objects can not be rendered.
- `PhysicsWorld`: implements physics simulation. Physics components such as `RigidBody` or `CollisionShape` can not function properly without this.
- `DebugRenderer`: implements debug geometry rendering.

Ordinary components like `Light`, `Camera` or `StaticModel` should not be created directly into the `Scene`, but rather into child nodes.

The library comes with a wide variety of components that you can attach to your nodes to bring them to life: user-visible elements (models), sounds, rigid bodies, collision shapes, cameras, light sources, particle emitters and much more.

Shapes

As a convenience, various shapes are available as simple nodes in the Urho.Shapes namespace. These include boxes, spheres, cones, cylinders and planes.

Camera and Viewport

Just like the light, cameras are components, so you will need to attach the component to a node, this is done like this:

```
var CameraNode = scene.CreateChild ("camera");
camera = CameraNode.CreateComponent<Camera>();
CameraNode.Position = new Vector3 (0, 5, 0);
```

With this, you have created a camera, and you have placed the camera in the 3D world, the next step is to inform the `Application` that this is the camera that you want to use, this is done with the following code:

```
Renderer.SetViewPort (0, new Viewport (Context, scene, camera, null))
```

And now you should be able to see the results of your creation.

Identification and scene hierarchy

Unlike nodes, components do not have names; components inside the same node are only identified by their type, and index in the node's component list, which is filled in creation order, for example, you can retrieve the `Light` component out of the `lightNode` object above like this:

```
var myLight = lightNode.GetComponent<Light>();
```

You can also get a list of all the components by retrieving the `Components` property which returns an `IList<Component>` that you can use.

When created, both nodes and components get scene-global integer IDs. They can be queried from the Scene by using the functions `GetNode(uint id)` and `GetComponent(uint id)`. This is much faster than for example doing recursive name-based scene node queries.

There is no built-in concept of an entity or a game object; rather it is up to the programmer to decide the node hierarchy, and in which nodes to place any scripted logic. Typically, free-moving objects in the 3D world would be created as children of the root node. Nodes can be created either with or without a name using `CreateChild()`. Uniqueness of node names is not enforced.

Whenever there is some hierarchical composition, it is recommended (and in fact necessary, because components do not have their own 3D transforms) to create a child node.

For example if a character was holding an object in his hand, the object should have its own node, which would be parented to the character's hand bone (also a `Node`). The exception is the physics `CollisionShape`, which can be offsetted and rotated individually in relation to the node.

Note that `Scene`'s own transform is purposefully ignored as an optimization when calculating world derived transforms of child nodes, so changing it has no effect and it should be left as it is (position at origin, no rotation, no scaling.)

`Scene` nodes can be freely reparented. In contrast components always belong to the node they attached to, and can not be moved between nodes. Both nodes and components provide a `Remove()` function to accomplish this without having to go through the parent. Once the node is removed, no operations on the node or component in question are safe after calling that function.

It is also possible to create a `Node` that does not belong to a scene. This is useful for example with a camera moving in a scene that may be loaded or saved, because then the camera will not be saved along with the actual scene, and will not be destroyed when the scene is loaded. However, note that creating geometry, physics or script components to an unattached node, and then moving it into a scene later will cause those components to not work correctly.

Scene updates

A Scene whose updates are enabled (default) will be automatically updated on each main loop iteration. The application's `SceneUpdate` event handler is invoked on it.

Nodes and components can be excluded from the scene update by disabling them, see `Enabled`. The behavior depends on the specific component, but for example disabling a drawable component also makes it invisible, while disabling a sound source component mutes it. If a node is disabled, all of its components are treated as disabled regardless of their own enable/disable state.

Adding Behavior to Your Components

The best way to structure your game is to make your own component that encapsulate an actor or element on your game. This makes the feature self contained, from the assets used to display it, to its behavior.

The simplest way of adding behavior to a component is to use actions, which are instructions that you can queue and combine that with C# async programming. This allows the behavior for your component to be very high level and makes it simpler to understand what is happening.

Alternatively, you can control exactly what happens to your component by updating your component properties

on each frame (discussed in Frame-based Behavior section).

Actions

You can add behavior to nodes very easily using Actions. Actions can alter various node properties and execute them over a period of time, or repeat them a number of times with a given animation curve.

For example, consider a "cloud" node on your scene, you can fade it like this:

```
await cloud.RunActionsAsync (new FadeOut (duration: 3))
```

Actions are immutable objects, which allows you to reuse the action for driving different objects.

A common idiom is to create an action that performs the reverse operation:

```
var gotoExit = new MoveTo (duration: 3, position: exitLocation);
var return = gotoExit.Reverse ();
```

The following example would fade the object for you over a period of three seconds. You can also run one action after another, for example, you could first move the cloud, and then hide it:

```
await cloud.RunActionsAsync (
    new MoveBy  (duration: 1.5f, position: new Vector3(0, 0, 15),
    new FadeOut (duration: 3));
```

If you want both actions to take place at the same time, you can use the Parallel action, and provide all the actions you want done in parallel:

```
  await cloud.RunActionsAsync (
    new Parallel (
      new MoveBy  (duration: 3, position: new Vector3(0, 0, 15),
      new FadeOut (duration: 3)));
```

In the above example the cloud will move and fade out at the same time.

You will notice that these are using C# await, which allows you to think linearly about the behavior you want to achieve.

Basic Actions

These are the actions supported in UrhoSharp:

- Moving nodes: `MoveTo`, `MoveBy`, `Place`, `BezierTo`, `BezierBy`, `JumpTo`, `JumpBy`
- Rotating nodes: `RotateTo`, `RotateBy`
- Scaling nodes: `ScaleTo`, `ScaleBy`
- Fading nodes: `FadeIn`, `FadeTo`, `FadeOut`, `Hide`, `Blink`
- Tinting: `TintTo`, `TintBy`
- Instants: `Hide`, `Show`, `Place`, `RemoveSelf`, `ToggleVisibility`
- Looping: `Repeat`, `RepeatForever`, `ReverseTime`

Other advanced features include the combination of the `Spawn` and `Sequence` actions.

Easing - Controlling the Speed of Your Actions

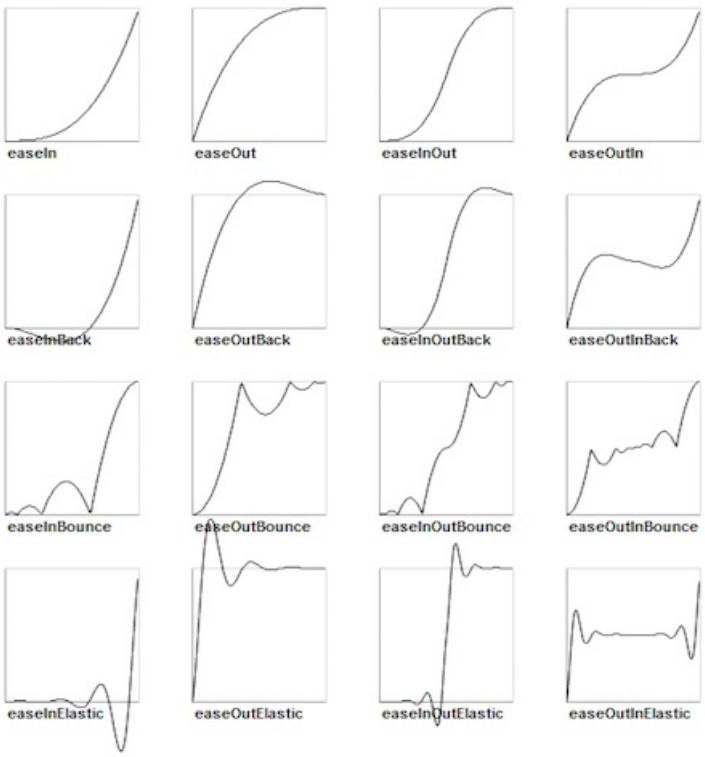
Easing is a way that directs the way that the animation will unfold, and it can make your animations a lot more pleasant. By default your actions will have a linear behavior, for example a `MoveTo` action would have a very robotic movement. You can wrap your Actions into an Easing action to change the behavior, for example, one that

would slowly start the movement, accelerate and slowly reach the end (`EasyInOut`).

You do this by wrapping an existing Action into an easing action, for example:

```
await cloud.RunActionAsync (
   new EaseInOut (
     new MoveTo (duration: 3, position: new Vector (0,0,15)), rate:1))
```

There are many easing modes, the following chart shows the various easing types and their behavior on the value of the object they are controlling over the period of time, from start to finish:



Using Actions and Async Code

In your `Component` subclass, you should introduce an async method that prepares your component behavior and drives the functionality for it. Then you would invoke this method using the C# `await` keyword from another part of your program, either your `Application.Start` method or in response to a user or story point in your application.

For example:

```
class Robot : Component {
    public bool IsAlive;
    async void Launch ()
    {
        // Dress up our robot
        var cache = Application.ResourceCache;
        var model = node.CreateComponent<StaticModel>();
        model.Model = cache.GetModel ("robot.mdl"));
        model.SetMaterial (cache.GetMaterial ("robot.xml"));
        Node.SetScale (1);

        // Bring the robot into our scene
        await Node.RunActionsAsync(
            new MoveBy(duration: 0.6f, position: new Vector3(0, -2, 0)));
        // Make him move around to avoid the user fire
        MoveRandomly(minX: 1, maxX: 2, minY: -3, maxY: 3, duration: 1.5f);
        // And simultaneously have him shoot at the user
        StartShooting();
    }

    protected async void MoveRandomly (float minX, float maxX,
                                       float minY, float maxY,
                       float duration)
    {
        while (IsAlive){
            var moveAction = new MoveBy(duration,
                new Vector3(RandomHelper.NextRandom(minX, maxX),
                            RandomHelper.NextRandom(minY, maxY), 0));
            await Node.RunActionsAsync(moveAction, moveAction.Reverse());
        }
    }
    protected async void StartShooting()
    {
        while (IsAlive && Node.Components.Count > 0){
            foreach (var weapon in Node.Components.OfType<Weapon>()){
                await weapon.FireAsync(false);
                if (!IsAlive)
                    return;
            }
            await Node.RunActionsAsync(new DelayTime(0.1f));
        }
    }
}
```

In the `Launch` method above three actions are started: the robot comes into the scene, this action will alter the location of the node over a period of 0.6 seconds. Since this is an async option, this will happen concurrently as the next instruction which is the call to `MoveRandomly`. This method will alter the position of the robot in parallel to a random location. This is achieved by performing two compounded actions, the movement to a new location, and going back to the original position and repeat this as long as the robot remains alive. And to make things more interesting, the robot will keep shooting simultaneously. The shooting will only start every 0.1 seconds.

Frame-based Behavior Programming

If you want to control the behavior of your component on a frame-by-frame basis instead of using actions, what you would do is to override the `OnUpdate` method of your `Component` subclass. This method is invoked once per frame, and is invoked only if you set the ReceiveSceneUpdates property to true.

The following shows how to create a `Rotator` component, that is then attached to a Node, which causes the node to rotate:

```
class Rotator : Component {
    public Rotator()
    {
        ReceiveSceneUpdates = true;
    }
    public Vector3 RotationSpeed { get; set; }
    protected override void OnUpdate(float timeStep)
    {
        Node.Rotate(new Quaternion(
            RotationSpeed.X * timeStep,
            RotationSpeed.Y * timeStep,
            RotationSpeed.Z * timeStep),
            TransformSpace.Local);
    }
}
```

And this is how you would attach this component to a node:

```
Node boxNode = new Node();
var rotator = new Rotator() { RotationSpeed = rotationSpeed };
boxNode.AddComponent (rotator);
```

Combining Styles

You can use the async/action based model for programming much of the behavior which is great for fire-and-forget style of programming, but you can also fine tune your component's behavior to also run some update code on each frame.

For example, in the SamplyGame demo this is used in the `Enemy` class encodes the basic behavior uses actions, but it also ensures that the components point toward the user by setting direction of the node with `Node.LookAt`:

```
    protected override void OnUpdate(SceneUpdateEventArgs args)
    {
        Node.LookAt(
            new Vector3(0, -3, 0),
            new Vector3(0, 1, -1),
            TransformSpace.World);
        base.OnUpdate(args);
    }
```

Loading and saving scenes

Scenes can be loaded and saved in XML format; see the functions `LoadXml` and `SaveXML()`. When a scene is loaded, all existing content in it (child nodes and components) is removed first. Nodes and components that are marked temporary with the `Temporary` property will not be saved. The serializer handles all built-in components and properties but it's not smart enough to handle custom properties and fields defined in your Component subclasses. However it provides two virtual methods for this:

- `OnSerialize` where you can register you custom states for the serialization
- `OnDeserialized` where you can obtain your saved custom states.

Typically, a custom component will look like the following:

```
class MyComponent : Component {
 // Constructor needed for deserialization
 public MyComponent(IntPtr handle) : base(handle) { }
 public MyComponent() { }
 // user defined properties (managed state):
 public Quaternion MyRotation { get; set; }
 public string MyName { get; set; }

 public override void OnSerialize(IComponentSerializer ser)
 {
  // register our properties with their names as keys using nameof()
  ser.Serialize(nameof(MyRotation), MyRotation);
  ser.Serialize(nameof(MyName), MyName);
 }

 public override void OnDeserialize(IComponentDeserializer des)
 {
  MyRotation = des.Deserialize<Quaternion>(nameof(MyRotation));
  MyName = des.Deserialize<string>(nameof(MyName));
 }
 // called when the component is attached to some node
 public override void OnAttachedToNode()
 {
  var node = this.Node;
 }
}
```

Object Prefabs

Just loading or saving whole scenes is not flexible enough for games where new objects need to be dynamically created. On the other hand, creating complex objects and setting their properties in code will also be tedious. For this reason, it is also possible to save a scene node which will include its child nodes, components and attributes. These can later conveniently be loaded as a group. Such a saved object is often referred to as a prefab. There are three ways to do this:

- In code by calling `Node.SaveXml` on the Node
- In the editor, by selecting the node in the hierarchy window and choosing "Save node as" from the "File" menu.
- Using the "node" command in `AssetImporter`, which will save the scene node hierarchy and any models contained in the input asset (eg. a Collada file)

To instantiate the saved node into a scene, call `InstantiateXml()`. The node will be created as a child of the Scene but can be freely reparented after that. Position and rotation for placing the node need to be specified. The following code demonstrates how to instantiate a prefab `Ninja.xm` to a scene with desired position and rotation:

```
var prefabPath = Path.Combine (FileSystem.ProgramDir,"Data/Objects/Ninja.xml");
using (var file = new File(Context, prefabPath, FileMode.Read))
{
    scene.InstantiateXml(file, desiredPos, desiredRotation,
        CreateMode.Replicated);
}
```

Events

UrhoObjects raise a number of events, these are surfaced as C# events on the various classes that generate them. In addition to the C#-based event model, it is also possible to use a the `SubscribeToXXX` methods that will allow you to subscribe and keep a subscription token that you can later use to unsubscribe. The difference is that the former will allow many callers to subscribe, while the second one only allows one, but allows for the nicer lambda-style approach to be used, and yet, allow for easy removal of the subscription. They are mutually exclusive.

When you subscribe to an event, you must provide a method that takes an argument with the appropriate event arguments.

For example, this is how you subscribe to a mouse button down event:

```
public void override Start ()
{
    UI.MouseButtonDown += HandleMouseButtonDown;
}

void HandleMouseButtonDown(MouseButtonDownEventArgs args)
{
 Console.WriteLine ("button pressed");
}
```

With lambda style:

```
public void override Start ()
{
    UI.MouseButtonDown += args => {
        Console.WriteLine ("button pressed");
    };
}
```

Sometimes you will want to stop receiving notifications for the event, in those cases, save the return value from the call to `SubscribeTo` method, and invoke the Unsubscribe method on it:

```
Subscription mouseSub;

public void override Start ()
{
    mouseSub = UI.SubscribeToMouseButtonDown (args => {
 Console.WriteLine ("button pressed");
      mouseSub.Unsubscribe ();
    };
}
```

The parameter received by the event handler is a strongly typed event arguments class that will be specific to each event and contains the event payload.

Responding to User Input

You can subscribe to various events, like keystrokes down by subscribing to the event, and responding to the input being delivered:

```
Start ()
{
    UI.KeyDown += HandleKeyDown;
}

void HandleKeyDown (KeyDownEventArgs arg)
{
     switch (arg.Key){
     case Key.Esc: Engine.Exit (); return;
}
```

But in many scenarios, you want your scene update handlers to check on the current status of the keys when they are being updated, and update your code accordingly. For example, the following can be used to update the

camera location based on the keyboard input:

```
protected override void OnUpdate(float timeStep)
{
    Input input = Input;
    // Movement speed as world units per second
    const float moveSpeed = 4.0f;
    // Read WASD keys and move the camera scene node to the
    // corresponding direction if they are pressed
    if (input.GetKeyDown(Key.W))
        CameraNode.Translate(Vector3.UnitY * moveSpeed * timeStep, TransformSpace.Local);
    if (input.GetKeyDown(Key.S))
        CameraNode.Translate(new Vector3(0.0f, -1.0f, 0.0f) * moveSpeed * timeStep, TransformSpace.Local);
    if (input.GetKeyDown(Key.A))
        CameraNode.Translate(new Vector3(-1.0f, 0.0f, 0.0f) * moveSpeed * timeStep, TransformSpace.Local);
    if (input.GetKeyDown(Key.D))
        CameraNode.Translate(Vector3.UnitX * moveSpeed * timeStep, TransformSpace.Local);
}
```

Resources (Assets)

Resources include most things in UrhoSharp that are loaded from mass storage during initialization or runtime:

- `Animation` - used for skeletal animations
- `Image` - represents images stored in a variety of graphic formats
- `Model` - 3D Models
- `Material` - materials used to render Models.
- `ParticleEffect` - describes how a particle emitter works, see "Particles" below.
- `Shader` - custom shaders
- `Sound` - sounds to playback, see "Sound" below.
- `Technique` - material rendering techniques
- `Texture2D` - 2D texture
- `Texture3D` - 3D texture
- `TextureCube` - Cube texture
- `XmlFile`

They are managed and loaded by the `ResourceCache` subsystem (available as `Application.ResourceCache`).

The resources themselves are identified by their file paths, relative to the registered resource directories or package files. By default, the engine registers the resource directories `Data` and `CoreData`, or the packages `Data.pak` and `CoreData.pak` if they exist.

If loading a resource fails, an error will be logged and a null reference is returned.

The following example shows a typical way of fetching a resource from the resource cache. In this case, a texture for a UI element, this uses the `ResourceCache` property from the `Application` class.

```
healthBar.SetTexture(ResourceCache.GetTexture2D("Textures/HealthBarBorder.png"));
```

Resources can also be created manually and stored to the resource cache as if they had been loaded from disk.

Memory budgets can be set per resource type: if resources consume more memory than allowed, the oldest resources will be removed from the cache if not in use anymore. By default the memory budgets are set to unlimited.

Bringing 3D-Models and Images

Urho3D tries to use existing file formats whenever possible, and define custom file formats only when absolutely necessary such as for models (*.mdl) and for animations (*.ani). For these types of assets, Urho provides a converter - AssetImporter which can consume many popular 3D formats such as fbx, dae, 3ds, and obj, etc.

There is also a handy add-in for Blender https://github.com/reattiva/Urho3D-Blender that can export your Blender assets in the format that is suitable for Urho3D.

Background loading of resources

Normally, when requesting resources using one of the `ResourceCache`'s `Get` method, they are loaded immediately in the main thread, which may take several milliseconds for all the required steps (load file from disk, parse data, upload to GPU if necessary) and can therefore result in framerate drops.

If you know in advance what resources you need, you can request them to be loaded in a background thread by calling `BackgroundLoadResource()`. You can subscribe to the Resource Background Loaded event by using the `SubscribeToResourceBackgroundLoaded` method. it will tell if the loading actually was a success or a failure. Depending on the resource, only a part of the loading process may be moved to a background thread, for example the finishing GPU upload step always needs to happen in the main thread. Note that if you call one of the resource loading methods for a resource that is queued for background loading, the main thread will stall until its loading is complete.

The asynchronous scene loading functionality `LoadAsync()` and `LoadAsyncXML()` has the option to background load the resources first before proceeding to load the scene content. It can also be used to only load the resources without modifying the scene, by specifying the `LoadMode.ResourcesOnly`. This allows to prepare a scene or object prefab file for fast instantiation.

Finally the maximum time (in milliseconds) spent each frame on finishing background loaded resources can be configured by setting the `FinishBackgroundResourcesMs` property on the `ResourceCache`.

Sound

Sound is an important part of game play, and the UrhoSharp framework provides a way of playing sounds in your game. You play sounds by attaching a `SoundSource` component to a `Node` and then playing a named file from your resources.

This is how it is done:

```
var explosionNode = Scene.CreateChild();
var sound = explosionNode.CreateComponent<SoundSource>();
soundSource.Play(Application.ResourceCache.GetSound("Sounds/ding.wav"));
soundSource.Gain = 0.5f;
soundSource.AutoRemove = true;
```

Particles

Particles provide a simple way of adding some simple and inexpensive effects to your application. You can consume particles stored in PEX format, using tools like http://onebyonedesign.com/flash/particleeditor/.

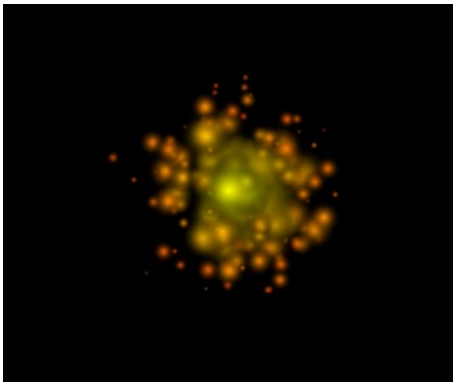
Particles are components that can be added to a node. You need to call the node's `CreateComponent<ParticleEmitter2D>` method to create the particle and then configure the particle by setting the Effect property to a 2D effect that is loaded from the resource cache.

For example, you can invoke this method on your component to show some particles that are rendered as an explosion when it hits:

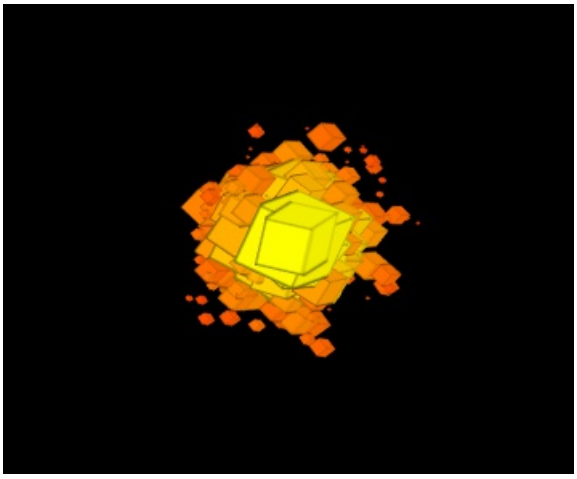
```
public async void Explode (Component target)
{
    // show a small explosion when the missile reaches an aircraft.
    var cache = Application.ResourceCache;
    var explosionNode = Scene.CreateChild();
    explosionNode.Position = target.Node.WorldPosition;
    explosionNode.SetScale(1f);
    var particle = explosionNode.CreateComponent<ParticleEmitter2D>();
    particle.Effect = cache.GetParticleEffect2D("explosion.pex");
    var scaleAction = new ScaleTo(0.5f, 0f);
    await explosionNode.RunActionsAsync(
        scaleAction, new DelayTime(0.5f));
    explosionNode.Remove();
}
```

The above code will create an explosion node that is attached to your current component, inside this explosion node we create a 2D particle emitter and configure it by setting the Effect property. We run two actions, one that scales the node to be smaller, and one that leaves it at that size for 0.5 seconds. Then we remove the explosion, which also removes the particle effect from the screen.

The above particle renders like this when using a sphere texture:



And this is what it looks if you use a blocky texture:



Multithreading Support

UrhoSharp is a single threaded library. This means that you should not attempt to invoke methods in UrhoSharp from a background thread, or you risk corrupting the application state, and likely crash your application.

If you want to run some code in the background and then update Urho components on the main UI, you can use the `Application.InvokeOnMain(Action)` method. Additionally, you can use C# await and the .NET task APIs to ensure that the code is executed on the proper thread.

UrhoEditor

You can download the Urho Editor for your platform from the Urho Website, go to Downloads and pick the latest version.

Copyrights

UrhoSharp Platform Support

10/3/2018 • 2 minutes to read • Edit Online

In this section, we discuss how to add Urho to an existing native project for your platform and how to take advantage of platform specific integration.

Android

Android setup instructions and features.

iOS and tvOS

iOS and tvOS setup instructions and features.

macOS

macOS setup instructions and features.

Windows

Windows setup instructions and features.

Xamarin.Forms

Xamarin.Forms setup instructions and sample.

UrhoSharp Android Support

10/3/2018 • 2 minutes to read • Edit Online

Android Specific Setup and Features

While Urho is a portable class library, and allows the same API to be used across the various platform for your game logic, you still need to initialize Urho in your platform specific driver, and in some cases, you will want to take advantage of platform specific features.

In the pages below, assume that `MyGame` is a subclass of the `Application` class.

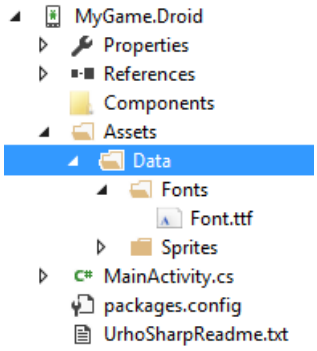
Architectures

Supported architectures: x86, armeabi, armeabi-v7a

Create a Project

Create an Android project, and add the UrhoSharp NuGet package.

Add Data containing your assets to the **Assets** directory and make sure all files have **AndroidAsset** as the **Build Action**.



Configure and Launching Urho

Add using statements for the `Urho` and `Urho.Android` namespaces, and then add this code for initializing Urho, as well as launching your application.

The simplest way to run a game, as implemented in the MyGame class is to call

```
UrhoSurface.RunInActivity<MyGame>();
```

This will open a fullscreen activity with the game as a content.

Custom Embedding of Urho

You can alternatively to having Urho take over the entire application screen, and to use it as a component of your application, you can create a `SurfaceView` via:

```
var surface = UrhoSurface.CreateSurface<MyGame>(activity)
```

You will also need to forward a few events from you activity to UrhoSharp, e.g:

```
protected override void OnPause()
{
 UrhoSurface.OnPause();
 base.OnPause();
}
```

You must do the same for: `OnResume`, `OnPause`, `OnLowMemory`, `OnDestroy`, `DispatchKeyEvent` and `OnWindowFocusChanged`.

This shows a typical Activity that launches the game:

```
[Activity(Label = "MyUrhoApp", MainLauncher = true,
    Icon = "@drawable/icon", Theme = "@android:style/Theme.NoTitleBar.Fullscreen",
    ConfigurationChanges = ConfigChanges.KeyboardHidden | ConfigChanges.Orientation,
    ScreenOrientation = ScreenOrientation.Portrait)]
public class MainActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        var mLayout = new AbsoluteLayout(this);
        var surface = UrhoSurface.CreateSurface<MyUrhoApp>(this);
        mLayout.AddView(surface);
        SetContentView(mLayout);
    }

    protected override void OnResume()
    {
        UrhoSurface.OnResume();
        base.OnResume();
    }

    protected override void OnPause()
    {
        UrhoSurface.OnPause();
        base.OnPause();
    }

    public override void OnLowMemory()
    {
        UrhoSurface.OnLowMemory();
        base.OnLowMemory();
    }

    protected override void OnDestroy()
    {
        UrhoSurface.OnDestroy();
        base.OnDestroy();
    }

    public override bool DispatchKeyEvent(KeyEvent e)
    {
        if (!UrhoSurface.DispatchKeyEvent(e))
            return false;
        return base.DispatchKeyEvent(e);
    }

    public override void OnWindowFocusChanged(bool hasFocus)
    {
        UrhoSurface.OnWindowFocusChanged(hasFocus);
        base.OnWindowFocusChanged(hasFocus);
    }
}
```

UrhoSharp iOS and tvOS Support

10/3/2018 • 2 minutes to read • Edit Online

While Urho is a portable class library, and allows the same API to be used across the various platform for your game logic, you still need to initialize Urho in your platform specific driver, and in some cases, you will want to take advantage of platform specific features.

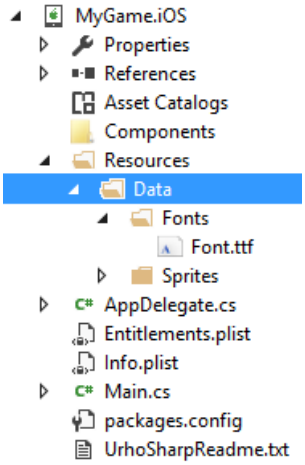
In the pages below, assume that `MyGame` is a sublcass of the `Application` class.

iOS and tvOS

Supported architectures: armv7, arm64, i386

Creating a Project

Create an iOS project, and then add Data to the Resources directory and make sure all files have **BundleResource** as the **Build Action**.



Configuring and Launching Urho

Add using statements for the `Urho` and `Urho.iOS` namespaces, and then add this code for initializing Urho, as well as launching your application:

```
new MyGame().Run();
```

Notice that since iOS expects `FinishedLaunching` to complete, you should queue the call to `Run()` to run after the method completes, this is a common idiom:

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    LaunchGame();
    return true;
}

async void LaunchGame()
{
    await Task.Yield();
    new SamplyGame().Run();
}
```

It is important that you disable PNG optimizations because the default iOS PNG optimizer will generate images that Urho can not currently properly consume

Custom Embedding of Urho

You can alternatively to having Urho take over the entire application screen, and to use it as a component of your application, you can create a `UrhoSurface` which is a `UIView` that you can embed in your existing application.

This is what you would need to do:

```
var view = new UrhoSurface () {
    Frame = new CGRect (100,100,200,200),
    BackgroundColor = UIColor.Red
}
window.AddSubview (view);
```

This will host your Urho class, so then you would do:

```
new MyGame().Run ();
```

UrhoSharp Mac Support

10/3/2018 • 2 minutes to read • Edit Online

Mac Specific Setup and Features

While Urho is a portable class library, and allows the same API to be used across the various platform for your game logic, you still need to initialize Urho in your platform specific driver, and in some cases, you will want to take advantage of platform specific features.

In the pages below, assume that `MyGame` is a subclass of the `Application` class.

macOS

Supported architectures: x86/x86-64 for 32 bit and 64 bit.

Creating a Project

Create a console project, reference the Urho NuGet and then make sure that you can locate the assets (the directories containing the Data directory).

```
DesktopUrhoInitializer.AssetsDirectory = "../Assets";
new MyGame().Run();
```

Example

Complete example

UrhoSharp Windows Support

11/11/2018 • 2 minutes to read • Edit Online

While Urho is a portable class library, and allows the same API to be used across the various platform for your game logic, you still need to initialize Urho in your platform specific driver, and in some cases, you will want to take advantage of platform specific features.

In the pages below, assume that `MyGame` is a subclass of the `Application` class.

Supported architectures: only 64bit Windows.

You can see complete examples showing how to use this in our samples

Standalone Project

Creating a Project

Create a Console project, reference the Urho NuGet and then make sure that you can locate the assets (the directories containing the Data directory).

Configuring and Launching Urho

To launch your application, do this:

```
DesktopUrhoInitializer.AssetsDirectory = "../Assets";
new MyGame().Run();
```

Example

Complete example

Integrated with WPF

Creating a Project

Create a WPF project, reference the Urho NuGet and then make sure that you can locate the assets (the directories containing the Data directory).

Configuring and Launching Urho from WPF

Create a subclass of `Window` and configure your assets like this:

```
public partial class MainWindow : Window
{
 Application currentApplication;

 public MainWindow()
 {
  InitializeComponent();
  DesktopUrhoInitializer.AssetsDirectory = @"../../Assets";
  Loaded += (s,e) => RunGame (new MyGame ());
 }

 async void RunGame(MyGame game)
 {
  var urhoSurface = new Panel { Dock = DockStyle.Fill };
  WindowsFormsHost.Child = urhoSurface;
  WindowsFormsHost.Focus();
  urhoSurface.Focus();
  await Task.Yield();
  var appOptions = new ApplicationOptions(assetsFolder: "Data")
   {
    ExternalWindow = RunInSdlWindow.IsChecked.Value ? IntPtr.Zero : urhoSurface.Handle,
    LimitFps = false, //true means "limit to 200fps"
   };
  currentApplication = Urho.Application.CreateInstance(value.Type, appOptions);
  currentApplication.Run();
 }
}
```

Example

Complete example

Integrated with UWP

Creating a Project

Create a UWP project, reference the Urho NuGet and then make sure that you can locate the assets (the directories containing the Data directory).

Configuring and Launching Urho from UWP

Create a subclass of `Window` and configure your assets like this:

```
{
   InitializeComponent();
   GameTypes = typeof(Sample).GetTypeInfo().Assembly.GetTypes()
    .Where(t => t.GetTypeInfo().IsSubclassOf(typeof(Application)) && t != typeof(Sample))
    .Select((t, i) => new TypeInfo(t, $"{i + 1}. {t.Name}", ""))
    .ToArray();
   DataContext = this;
   Loaded += (s, e) => RunGame (new MyGame ());
  }

  public void RunGame(TypeInfo value)
  {
   //at this moment, UWP supports assets only in pak files (see PackageTool)
   currentApplication = UrhoSurface.Run(value.Type, "Data.pak");
  }
 }
```

Example

Complete example

Integrated with Windows.Forms

Creating a Project

Create a Windows.Forms project, reference the Urho NuGet and then make sure that you can locate the assets (the directories containing the Data directory).

Configuring and Launching Urho from Windows.Forms

Launch Urho from your form, see Complete Sample

UrhoSharp Xamarin.Forms Support

10/3/2018 • 2 minutes to read • Edit Online

UrhoSharp can be hosted in Xamarin.Forms applications. The Urho code can be written in a PCL and shared across platforms.

Follow these instructions for adding UrhoSharp to Xamarin.Forms, and refer to the UrhoSharp documentation for information on how to construct 3D scenes and actions.

The UrhoSharp Xamarin.Forms sample shows a 3D interactive chart to demonstrate how to interact with UrhoSharp from Xamarin.Forms shared code.

Programming UrhoSharp with F#

10/3/2018 • 2 minutes to read • Edit Online

UrhoSharp can be programmed with F# using the same libraries and concepts used by C# programmers. The Using UrhoSharp article gives an overview of the UrhoSharp engine and should be read prior to this article.

Like many libraries that originated in the C++ world, many UrhoSharp functions return booleans or integers indicating success or failure. You should use `|> ignore` to ignore these values.

The sample program is a "Hello World" for UrhoSharp from F#.

Creating an empty project

There are no F# templates for UrhoSharp yet available, so to create your own UrhoSharp project you can either start with the sample or follow these steps:

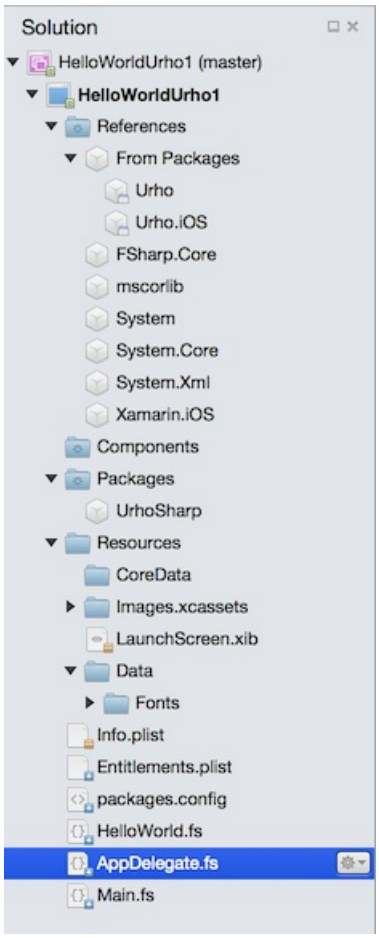
1. From Visual Studio for Mac, create a new **Solution**. Choose **iOS > App > Single View App** and select **F#** as the implementation language.
2. Delete the **Main.storyboard** file. Open the **Info.plist** file and in the **iPhone / iPod Deployment Info** pane, delete the `Main` string in the **Main Interface** dropdown.
3. Delete the **ViewController.fs** file as well.

Building Hello World in Urho

You are now ready to begin defining your game's classes. At a minimum, you will need to define a subclass of `Urho.Application` and override its `Start` method. To create this file, right-click on your F# project, choose **Add new file...** and add an empty F# class to your project. The new file will be added to the end of the list of files in your project, but you must drag it so that it appears *before* it is used in **AppDelegate.fs**.

1. Add a reference to the Urho NuGet package.
2. From an existing Urho project, copy the (large) directories **CoreData/** and **Data/** into your project's **Resources/** directory. In your F# project, right-click on the **Resources** folder and use **Add / Add Existing Folder** to add all of these files to your project.

Your project structure should now look like:



Define your newly-created class as a subtype of `Urho.Application` and override its `Start` method:

```
namespace HelloWorldUrho1

open Urho
open Urho.Gui
open Urho.iOS

type HelloWorld(o : ApplicationOptions) =
    inherit Urho.Application(o)

override this.Start() =
        let cache = this.ResourceCache
        let helloText = new Text()

        helloText.Value <- "Hello World from Urho3D, Mono, and F#"
        helloText.HorizontalAlignment <- HorizontalAlignment.Center
        helloText.VerticalAlignment <- VerticalAlignment.Center

        helloText.SetColor (new Color(0.f, 1.f, 0.f))
        let f = cache.GetFont("Fonts/Anonymous Pro.ttf")

        helloText.SetFont(f, 30) |> ignore

        this.UI.Root.AddChild(helloText)
```

The code is very straightforward. It uses the `Urho.Gui.Text` class to display a center-aligned string with a certain font and color size.

Before this code can run, though, UrhoSharp must be initialized.

Open the AppDelegate.fs file and modify the `FinishedLaunching` method as follows:

```
namespace HelloWorldUrho1

open System
open UIKit
open Foundation
open Urho
open Urho.iOS

[<Register ("AppDelegate")>]
type AppDelegate () =
    inherit UIApplicationDelegate ()

    override this.FinishedLaunching (app, options) =
        let o = ApplicationOptions.Default

        let g = new HelloWorld(o)
        g.Run() |> ignore

        true
```

The `ApplicationOptions.Default` provides the default options for a landscape-mode application. Pass these `ApplicationOptions` to the default constructor for your `Application` subclass (note that when you defined the `HelloWorld` class, the line `inherit Application(o)` calls the base-class constructor).

The `Run` method of your `Application` initiates the program. It is defined as returning an `int`, which can be piped to `ignore`.

The resulting program should look like:

Related Links

- Browse on GitHub (sample)

CocosSharp 2D Game Engine

10/3/2018 • 2 minutes to read • Edit Online

CocosSharp is a library for building 2D games using C# and F#. It is a .NET port of the popular Cocos2D engine.

Introduction to CocosSharp

The CocosSharp 2D game engine provides technology for making cross-platform games. For a full list of supported platforms see the CocosSharp wiki on GitHub. These guides use C# for code samples, although CocosSharp is fully functional with F# as well.

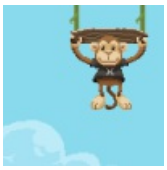
The core of CocosSharp is provided by the MonoGame framework, which is itself a cross-platform, hardware accelerated API providing graphics, audio, game state management, input, and a content pipeline for importing assets. CocosSharp is an efficient abstraction layer well suited for 2D games. Furthermore, larger games can perform their own optimizations outside of their core libraries as games grows in complexity. In other words, CocosSharp provides a mix of ease of use and performance, enabling developers to get started quickly without limiting game size or complexity.

This hands-on video shows how to create a simple cross-platform CocosSharp game:

BouncingGame

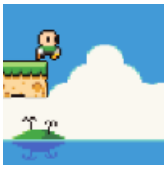
This guide describes BouncingGame, including how to work with game content, the various visual elements used to build a game, adding game logic, and more.

Fruity Falls game



This guide describes the Fruity Falls game, covering common CocosSharp and game development concepts such as physics, content management, game state, and game design.

Coin Time game



Coin Time is a full platformer game for iOS and Android. The goal of the game is to collect all of the coins in a

level and then reach the exit door while avoiding enemies and obstacles.

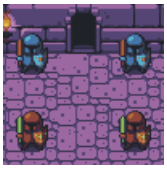
Drawing geometry with CCDrawNode

CCDrawNode provides methods for drawing primitive objects such as lines, circles, and triangles.

Animating with CCAction

`CCAction` is a base class that can be used to animate CocosSharp objects. This guide covers built-in `CCAction` implementations for common tasks such as positioning, scaling, and rotating. It also looks at how to create custom implementations by inheriting from `CCAction`.

Using Tiled with CocosSharp



Tiled is a powerful, flexible, and mature application for creating orthogonal and isometric tile maps for games. CocosSharp provides built-in integration for Tiled's native file format.

Entities in CocosSharp

The entity pattern is a powerful way to organize game code. It improves readability, makes code easier to maintain, and leverages built-in parent/child functionality.

Handling multiple resolutions in CocosSharp

This guide shows how to work with CocosSharp to develop games that display properly on devices of varying resolutions.

CocosSharp Content Pipeline

Content pipelines are often used in game development to optimize content and format it such that it can be loaded on certain hardware or with certain game development frameworks.

Improving frame rate with CCSpriteSheet

`CCSpriteSheet` provides functionality for combining and using many image files in one texture. Reducing texture count can improve a game's load times and framerate.

Texture caching using CCTextureCache

CocosSharp's `CCTextureCache` class provides a standard way to organize, cache, and unload content.

2D math with CocosSharp

This guide covers 2D mathematics for game development. It uses CocosSharp to show how to perform common game development tasks and explains the math behind these tasks.

Performance and visual effects with CCRenderTexture

The `CCRenderTexture` class provides functionality for rendering multiple CocosSharp objects to a single texture. Once created, `CCRenderTexture` instances can be used to render graphics efficiently and to implement visual effects.

BouncingGame details

11/11/2018 • 14 minutes to read • Edit Online

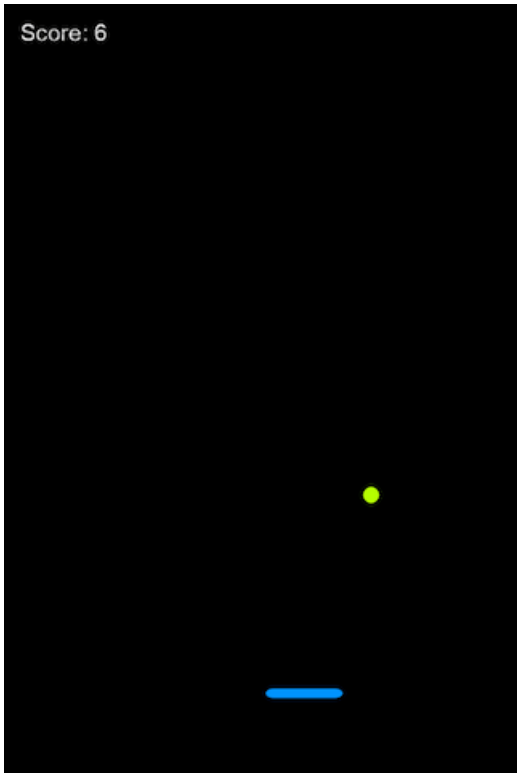
TIP

The code for BouncingGame can be found in the Xamarin samples. To best follow along with this guide, download and explore the code as the guide references it.

This walkthrough shows how to implement a simple bouncing ball game using CocosSharp.

- Unzipping game content
- Common CocosSharp visual elements
- Adding visual elements to `GameLayer`
- Implementing every-frame logic

Our finished game will look like this:



Unzipping game content

Game developers often use the term *content* to refer to non-code files which are usually created by visual artists, game designers, or audio designers. Common types of content include files used to display visuals, play sound, or control artificial intelligence (AI) behavior. From a game development team's perspective, content is usually created by non-programmers. Our game includes two types of content:

- png files to define how the ball and paddle appear.
- A single xnb file to define the font used by the score display (discussed in more detail when we add the score display below)

This content used here can be found in content zip. We'll need these files downloaded and unzipped to a location

that we will access later in this walkthrough.

Common CocosSharp visual elements

CocosSharp provides a number of classes used to display visuals. Some elements are directly visible, while others are used for organization. We'll use the following in the game:

- `CCNode` – Base class for all visual objects in CocosSharp. The `CCNode` class contains an `AddChild` function which can be used to construct a parent/child hierarchy, also referred to as a *visual tree* or *scene graph*. All classes mentioned below inherit from `CCNode`.
- `CCScene` – Root of the visual tree for all CocosSharp games. All visual elements must be part of a visual tree with a `CCScene` at the root, or they won't be visible.
- `CCLayer` – Container for visual objects, such as `CCSprite`. As the name implies, the `CCLayer` class is used to control how visual elements layer on top of each other.
- `CCSprite` – Displays an image or a portion of an image. `CCSprite` instances can be positioned, resized, and provide a number of visual effects.
- `CCLabel` – Displays a string on screen. The font used by `CCLabel` is defined in an xnb file. We'll discuss xnbs in more detail below.

To understand how the different types are used we'll consider a single `CCSprite`. Visual elements must be added to a `CCLayer`, and the visual tree must have a `CCScene` at its root. Therefore, the parent/child relationship for a single `CCSprite` would be `CCScene` > `CCLayer` > `CCSprite`.

Adding visual elements to GameLayer

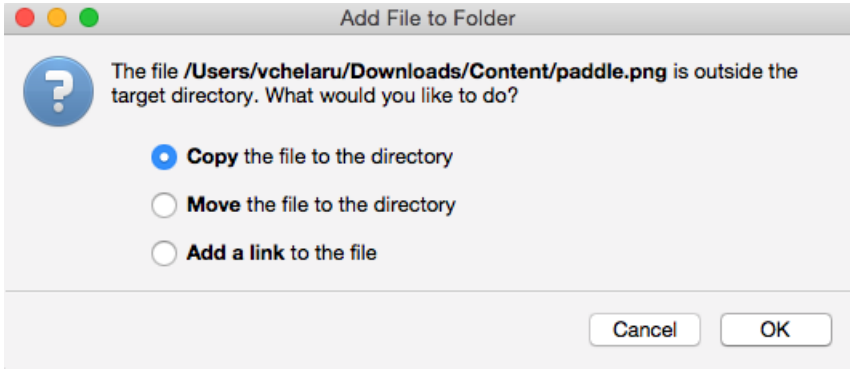
Adding the paddle sprite

Initially we'll set the game's background to black and also add a single `CCSprite` rendering on the screen. Modify the `GameLayer` class to add a `CCSprite` as follows:

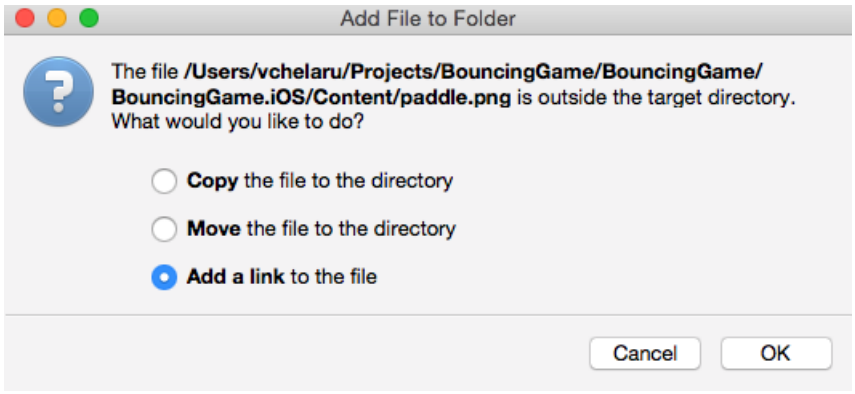
```
using System;
using System.Collections.Generic;
using CocosSharp;
namespace BouncingGame
{
    public class GameLayer : CCLayer
    {
        CCSprite paddleSprite;
        public GameLayer () : base(CCColor4B.Black)
        {
            // "paddle" refers to the paddle.png image
            paddleSprite = new CCSprite ("paddle");
            paddleSprite.PositionX = 100;
            paddleSprite.PositionY = 100;
            AddChild (paddleSprite);
        }
        protected override void AddedToScene ()
        {
            base.AddedToScene ();
            // Use the bounds to layout the positioning of the drawable assets
            CCRect bounds = VisibleBoundsWorldspace;
            // Register for touch events
            var touchListener = new CCEventListenerTouchAllAtOnce ();
            touchListener.OnTouchesEnded = OnTouchesEnded;
            AddEventListener (touchListener, this);
        }
        void OnTouchesEnded (List<CCTouch> touches, CCEvent touchEvent)
        {
            if (touches.Count > 0)
            {
                // Perform touch handling here
            }
        }
    }
}
```

The code above creates a single `CCSprite` and adds it as a child of the `GameLayer`. The `CCSprite` constructor allows us to define the image file to use as a string. Our code tells CocosSharp to look for a file called `paddle` (the extension is omitted, which we'll discuss later in this guide). CocosSharp will look for any file names `paddle` in the root content folder (which is **Content**) as well as any folders added to the `gameView.ContentManager.SearchPaths` (discussed in the previous section).

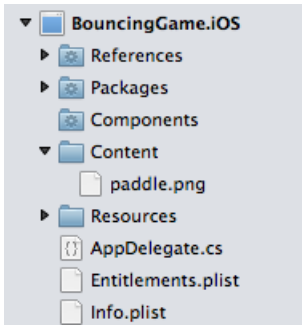
We'll add the files directly to the root **Content** folder for both iOS and Android. To do this, right-click or Control-click on the **Content** folder in the iOS project and select **Add** > **Add Files...** Navigate to where we unzipped the content earlier and select **paddle.png**. If asked about how to add the file to folder, we should select the **Copy** option:



Next, we'll add the file to the Android project. Right-click or Control-click on the Content folder (which is in the **Assets** folder on Android projects) and select **Add** > **Add Files...**. This time, navigate to the iOS project's **Content** folder. When asked about how to add the file, select the **Add a link** option:



We'll discuss why files had to be added to both projects below. Each project's **Content** folder now contain the **paddle.png** file:



If we run the game we'll see the `CCSprite` being drawn:

File details

So far we've added a single file to the project, and the process was fairly straight-forward. We simply added the **paddle.png** file to the **Content** folders and referenced it in code. Let's take a moment to look at some considerations when working with files in CocosSharp.

Capitalization

Notice that the file name and the string we used in code to access the file are both lower-case. This is because some platforms (such as Windows desktop and iOS simulator) are case insensitive, while other platforms (such as Android and iOS device) are case sensitive. We will use all lower-case files for the remainder of this tutorial so that the files and code remain as cross-platform as possible.

Extensions

The constructor for creating the Sprite does not include the ".png" extension when referencing the paddle file. The extension for files is typically omitted when working on CocosSharp projects as file extensions for the same asset type may differ between platforms. For example, audio files may be of .aiff, .mp3, or .wma file formats depending on the platform. Leaving off the extension allows the same code to work on all platforms regardless of the file

extension.

Content in platform-specific projects

Unlike most of the code files which can be in a PCL, content files must be added to each platform-specific project. CocosSharp requires this for two reasons:

1. Each platform has different **Build actions**. Content added to iOS projects should use the **BundleResource** build action. Content added to Android projects should use the **AndroidAsset** build action.
2. Some platforms require platform-specific file formats. For example, font .xnb files differ between iOS and Android, as we'll see later in this walkthrough.

If a file format does not differ between platforms (such as .png), then each platform can use the same file, where one or more platforms may link the file from the same location.

Orientation

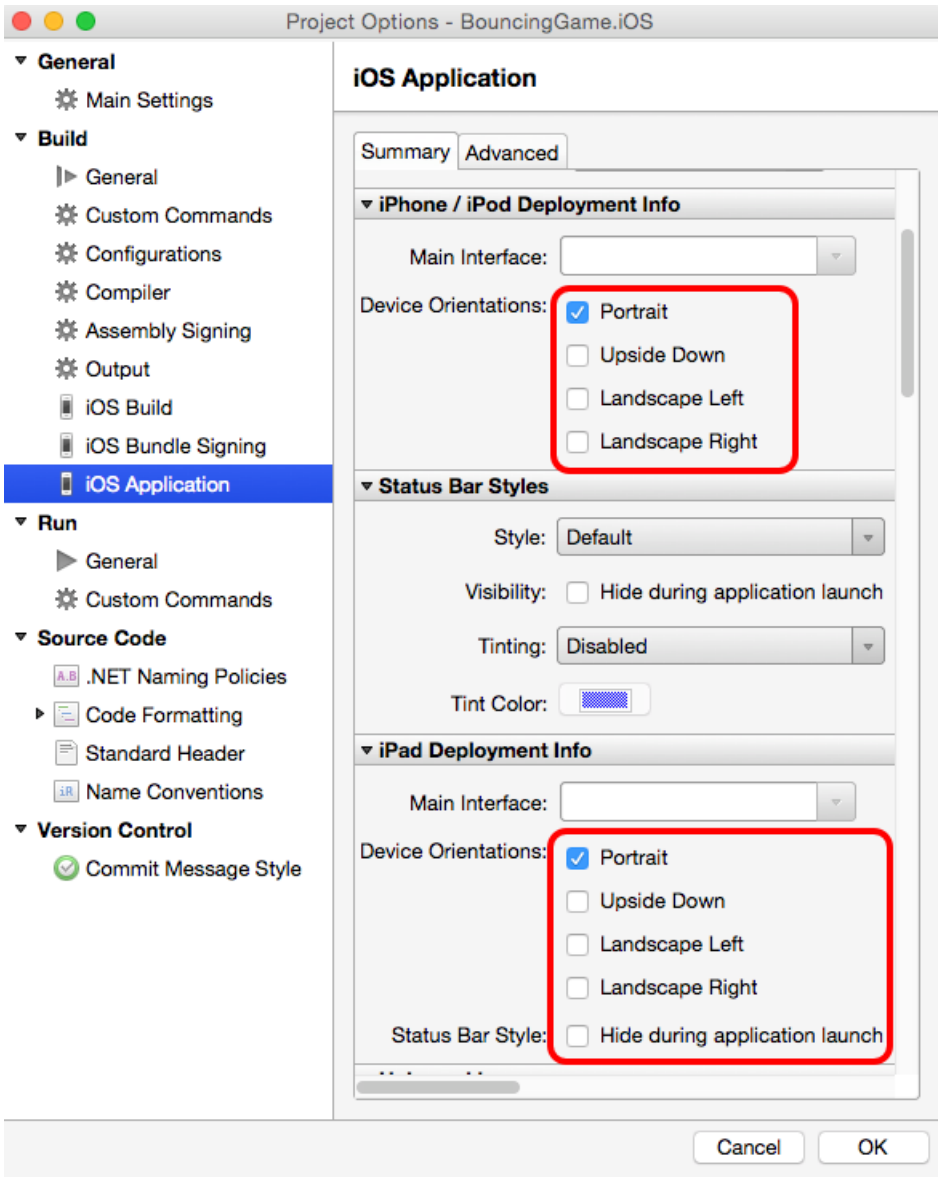
Just like any other app, CocosSharp apps can run in portrait or landscape orientations. We'll adjust the game to run in portrait-only mode. First, we'll change the resolution code in the game to handle a portrait aspect ratio. To do this, modify the `width` and `height` values in the `LoadGame` method in the `ViewController` class on iOS and `MainActivity` class on Android:

```
void LoadGame (object sender, EventArgs e)
{
    CCGameView gameView = sender as CCGameView;

    if (gameView != null)
    {
        var contentSearchPaths = new List<string> () { "Fonts", "Sounds" };
        CCSizeI viewSize = gameView.ViewSize;

        int width = 768;
        int height = 1027;
    // ...
```

Next we'll need to modify each platform-specific project to run in portrait mode.

iOS orientation

To modify the iOS project's orientation, Select the **Info.plist** file in the **BouncingGame.iOS** project, and change **iPhone/iPod Deployment Info** and the **iPad Deployment Info** to only include Portrait orientations:



Android orientation

To modify the Android project's orientation, open the MainActivity.cs file in the BouncingGame.Android project. Modify the Activity attribute definition so it specifies only a portrait orientation as follows:

```
[Activity (
    Label = "BouncingGame.Android",
    AlwaysRetainTaskState = true,
    Icon = "@drawable/icon",
    Theme = "@android:style/Theme.NoTitleBar",
    ScreenOrientation = ScreenOrientation.Portrait,
    LaunchMode = LaunchMode.SingleInstance,
    MainLauncher = true,
    ConfigurationChanges = ConfigChanges.Orientation | ConfigChanges.ScreenSize | ConfigChanges.Keyboard | Co
nfigChanges.KeyboardHidden)
]
public class MainActivity : Activity
{
...
```

Default coordinate system

Our code, which instantiates a `CCSprite`, sets the `PositionX` and `PositionY` values to 100. By default, this means that the `CCSprite` center will be positioned at 100 pixels up and to the right from the bottom-left of the screen. The coordinate system can be modified by attaching a `CCCamera` to the `CCLayer`. We won't be working with

`CCCamera` in this project, but more information on `CCCamera` can be found in the CocosSharp API docs.

The 100 pixels mentioned above "game" pixels as opposed to pixels on the hardware. This means that running the same game on a device of a different resolution (such as an iPad vs. an iPhone) will result in objects being positioned and sized correctly relative to the physical screen. Specifically, the game's visible area will always be 1024 pixels tall and 768 pixels wide, as this is the resolution we've specified earlier in the `LoadGame` method.

Adding the ball sprite

Now that we're familiar with the basics of working with `CCSprite`, we'll add the second `CCSprite` – a ball. We'll be following steps which are very similar to how we created the paddle `CCSprite`.

First, we'll add the **ball.png** image from the unzipped folder into the iOS project's **Content** folder. Select to **Copy** the file to the **Content** directory. Follow the same steps as above to add a link to the **ball.png** file in the Android project.

Next create the `CCSprite` for this ball by adding a member called `ballSprite` to the `GameScene` class, as well as instantiation code for the `ballSprite`. When finished the `GameLayer` class will look like this:

```
public class GameLayer : CCLayer
{
    CCSprite paddleSprite;
    CCSprite ballSprite;
    public GameLayer () : base (CCColor4B.Black)
    {
        // "paddle" refers to the paddle.png image
        paddleSprite = new CCSprite ("paddle");
        paddleSprite.PositionX = 100;
        paddleSprite.PositionY = 100;
        AddChild (paddleSprite);
        ballSprite = new CCSprite ("ball");
        ballSprite.PositionX = 320;
        ballSprite.PositionY = 600;
        AddChild (ballSprite);
    }
```

Adding the score label

The last visual element we'll add to the game is a `CCLabel` to display the number of times the user has successfully bounced the ball. The `CCLabel` uses a font specified in the constructor to display strings on screen.

Add the following code to create a `CCLabel` instance in `GameLayer`. Once finished the code should look like this:

```
public class GameLayer : CCLayer
{
    CCSprite paddleSprite;
    CCSprite ballSprite;
    CCLabel scoreLabel;
    public GameLayer () : base (CCColor4B.Black)
    {
        // "paddle" refers to the paddle.png image
        paddleSprite = new CCSprite ("paddle");
        paddleSprite.PositionX = 100;
        paddleSprite.PositionY = 100;
        AddChild (paddleSprite);
        ballSprite = new CCSprite ("ball");
        ballSprite.PositionX = 320;
        ballSprite.PositionY = 600;
        AddChild (ballSprite);
        scoreLabel = new CCLabel ("Score: 0", "Arial", 20, CCLabelFormat.SystemFont);
        scoreLabel.PositionX = 50;
        scoreLabel.PositionY = 1000;
        scoreLabel.AnchorPoint = CCPoint.AnchorUpperLeft;
        AddChild (scoreLabel);
    }
    // ...
```

Notice that the scoreLabel is using an `AnchorPoint` of `CCPoint.AnchorUpperLeft`, which means that the `PositionX` and `PositionY` values define its upper-left position. This lets us position the `scoreLabel` relative to the top-left of the screen without having to consider the label's dimensions.

Implementing every-frame logic

So far, the game presents a static scene. We'll be adding logic to control the movement of objects in the scene by adding code which updates the position of the objects at a high frequency. In this case, the code will run sixty times per second - also referred to as sixty *frames* per second (unless the hardware cannot handle updating this frequently). Specifically, we'll be adding logic to make the ball fall and bounce against the paddle, to move the paddle according to input, and to update the player's score every time the ball strikes the paddle.

The `Schedule` method, which is provided by the `CCNode` class, lets us add every-frame logic to the game. We'll add the code after `// New code` to the GameLayer constructor:

```
public GameLayer () : base (CCColor4B.Black)
{
    // "paddle" refers to the paddle.png image
    paddleSprite = new CCSprite ("paddle");
    paddleSprite.PositionX = 100;
    paddleSprite.PositionY = 100;
    AddChild (paddleSprite);
    ballSprite = new CCSprite ("ball");
    ballSprite.PositionX = 320;
    ballSprite.PositionY = 600;
    AddChild (ballSprite);
        scoreLabel = new CCLabel ("Score: 0", "arial", 22, CCLabelFormat.SpriteFont);
    scoreLabel.PositionX = 50;
    scoreLabel.PositionY = 1000;
    scoreLabel.AnchorPoint = CCPoint.AnchorUpperLeft;
    AddChild (scoreLabel);
    // New code:
    Schedule (RunGameLogic);
}
```

Now create a `RunGameLogic` method in the `GameLayer` class, which will house all of the every-frame logic:

```
void RunGameLogic(float frameTimeInSeconds)
{
}
```

The float parameter defines how long the frame is in seconds. We'll be using this value when implementing the ball's movement.

Making the ball fall

We can make the ball fall by either manually implementing gravity code or by using the built-in Box2D functionality in CocosSharp. The Box2D physics simulation engine is available to CocosSharp games. It is very powerful and efficient, but requires writing setup code. Since the physics simulation is fairly simple, here it will be implemented manually.

To implement gravity we'll need to store the current X and Y velocity of the ball. We'll add two members to the `GameLayer` class:

```
float ballXVelocity;
float ballYVelocity;
// How much to modify the ball's y velocity per second:
const float gravity = 140;
```

Next we can implement the falling logic in `RunGameLogic`:

```
void RunGameLogic(float frameTimeInSeconds)
{
    // This is a linear approximation, so not 100% accurate
    ballYVelocity += frameTimeInSeconds * -gravity;
    ballSprite.PositionX += ballXVelocity * frameTimeInSeconds;
    ballSprite.PositionY += ballYVelocity * frameTimeInSeconds;
}
```

Moving the paddle with touch input

Now that the ball is falling, we'll add horizontal movement to the paddle by using the `CCEventListenerTouchAllAtOnce` object. This object provides a number of input-related events. In this case, we want to be notified if any touch points move so we can adjust the position of the paddle. The `GameLayer` already instantiates a `CCEventListenerTouchAllAtOnce`, so we simply need to assign the `OnTouchesMoved` delegate. To do so, modify the AddedToScene method as follows:

```
protected override void AddedToScene ()
{
    base.AddedToScene ();
    // Use the bounds to layout the positioning of the drawable assets
    CCRect bounds = VisibleBoundsWorldspace;
    // Register for touch events
    var touchListener = new CCEventListenerTouchAllAtOnce ();
    touchListener.OnTouchesEnded = OnTouchesEnded;
    // new code:
    touchListener.OnTouchesMoved = HandleTouchesMoved;
    AddEventListener (touchListener, this);
}
```

Next, we'll implement `HandleTouchesMoved`:

```
void HandleTouchesMoved (System.Collections.Generic.List<CCTouch> touches, CCEvent touchEvent)
{
    // we only care about the first touch:
    var locationOnScreen = touches [0].Location;
    paddleSprite.PositionX = locationOnScreen.X;
}
```

Implementing ball collision

If we run the game now we'll notice that the ball falls through the paddle. We'll implement *collision* (logic to react to overlapping game objects) in the every-frame code. Since move objects change positions every frame, collision checking is usually also performed every frame. We'll also be adding velocity on the X axis when the ball hits the paddle to add some challenge to the game, but this means we need to keep the ball from moving past the edges of the screen. We'll do this in the `RunGameLogic` code after applying velocity to the `ballSprite` after `// New Code`:

```
void RunGameLogic(float frameTimeInSeconds)
{
    // This is a linear approximation, so not 100% accurate
    ballYVelocity += frameTimeInSeconds * -gravity;
    ballSprite.PositionX += ballXVelocity * frameTimeInSeconds;
    ballSprite.PositionY += ballYVelocity * frameTimeInSeconds;
    // New Code:
    // Check if the two CCSprites overlap...
    bool doesBallOverlapPaddle = ballSprite.BoundingBoxTransformedToParent.IntersectsRect(
        paddleSprite.BoundingBoxTransformedToParent);
    // ... and if the ball is moving downward.
    bool isMovingDownward = ballYVelocity < 0;
    if (doesBallOverlapPaddle && isMovingDownward)
    {
        // First let's invert the velocity:
        ballYVelocity *= -1;
        // Then let's assign a random value to the ball's x velocity:
        const float minXVelocity = -300;
        const float maxXVelocity = 300;
        ballXVelocity = CCRandom.GetRandomFloat (minXVelocity, maxXVelocity);
    }
    // First let’s get the ball position:
    float ballRight = ballSprite.BoundingBoxTransformedToParent.MaxX;
    float ballLeft = ballSprite.BoundingBoxTransformedToParent.MinX;
    // Then let’s get the screen edges
    float screenRight = VisibleBoundsWorldspace.MaxX;
    float screenLeft = VisibleBoundsWorldspace.MinX;
    // Check if the ball is either too far to the right or left:
    bool shouldReflectXVelocity =
        (ballRight > screenRight && ballXVelocity > 0) ||
        (ballLeft < screenLeft && ballXVelocity < 0);
    if (shouldReflectXVelocity)
    {
        ballXVelocity *= -1;
    }
}
```

Adding scoring

Now that the game is playable, the last step is to add scoring logic. First, we'll add a score member to the GameLayer class named `score`:

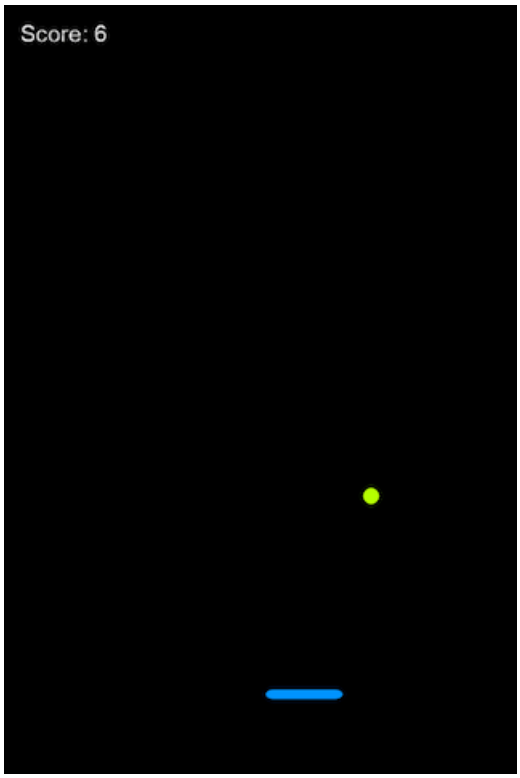
```
int score;
```

We'll use this variable to keep track of the player's score, and to display it using the `scoreLabel`. To do this add the

following code inside the if-statement in `RunGameLogic` when the ball and paddle overlap:

```
// ...
if (doesBallOverlapPaddle && isMovingDownward)
{
    // First let's invert the velocity:
    ballYVelocity *= -1;
    // Then let's assign a random to the ball's x velocity:
    const float minXVelocity = -300;
    const float maxXVelocity = 300;
    ballXVelocity = CCRandom.GetRandomFloat (minXVelocity, maxXVelocity);
    // New code:
    score++;
    scoreLabel.Text = "Score: " + score;
}
// ...
```

Now we can run the game and see that the game displays a score which increments as the ball bounces off of the paddle:



Summary

This walkthrough presented creating a cross-platform game with graphics, physics, and input using CocosSharp. It is the first step in getting started with CocosSharp game development. We looked at some of the most common classes in CocosSharp, how to construct a visual tree so objects rendered properly, and how to implement every-frame game logic.

This walkthrough covered only a small part of what the CocosSharp game engine offers. For information and walkthroughs on other CocosSharp topics, see the rest of the CocosSharp guides.

Related links

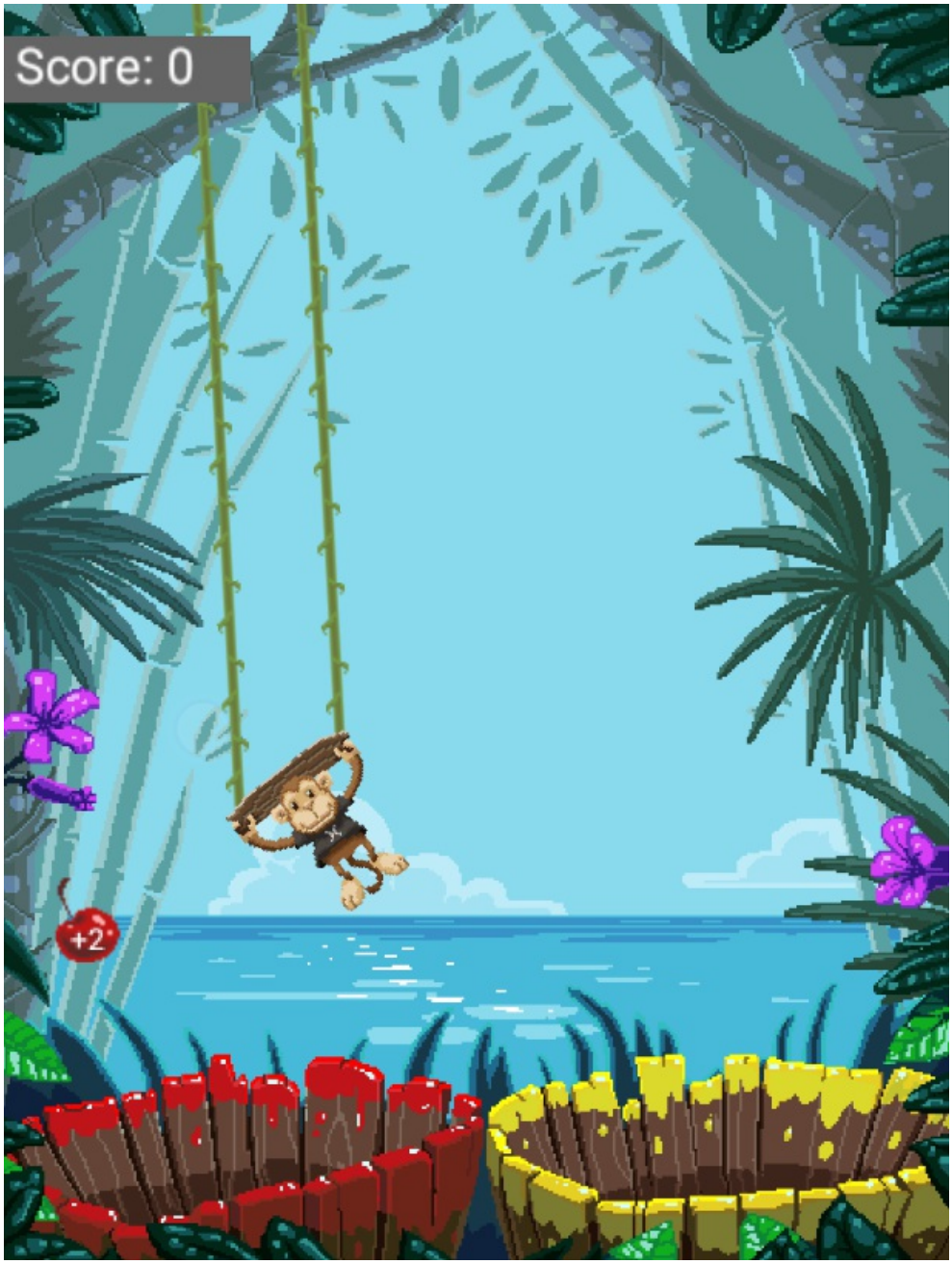
- Completed Game (sample)
- Game Content (sample)

Fruity Falls game details

10/3/2018 • 15 minutes to read • Edit Online

This guide reviews the Fruity Falls game, covering common CocosSharp and game development concepts such as physics, content management, game state, and game design.

Fruity Falls is a simple, physics-based game in which the player sorts red and yellow fruit into colored buckets to earn points. The goal of the game is to earn as many points as possible without letting a fruit drop into the wrong bin, ending the game.



Fruity Falls extends the concepts introduced in the BouncingGame guide by adding the following:

- Content in the form of PNGs
- Advanced physics
- Game state (transition between scenes)
- Ability to tune the game coefficients through variables contained in a single class

- Built-in visual debugging support
- Code organization using game entities
- Game design focused on fun and replay value

While the BouncingGame guide focused on introducing core CocosSharp concepts, Fruity Falls shows how to bring it all together into a finished game product. Since this guide references the BouncingGame, readers should first familiarize themselves with the BouncingGame guide prior to reading this guide.

This guide covers the implementation and design of Fruity Falls to provide insights to help you make your own game. It covers the following topics:

- GameController class
- Game entities
- Fruit graphics
- Physics
- Game content
- GameCoefficients

GameController class

The Fruity Falls PCL project includes a `GameController` class which is responsible for instantiating the game and moving between scenes. This class is used by both the iOS and Android projects to eliminate duplicate code.

The code contained within the `Initialize` method is similar to the code in the `Initialize` method in an unchanged CocosSharp template, but it contains a number of modifications.

By default, the `GameView.ContentManager.SearchPaths` depend on the device's resolution. As explained below in more detail, Fruity Falls uses the same content regardless of device resolution, so the `Initialize` method adds the `Images` path (with no subfolders) to the `SearchPaths`:

```
contentSearchPaths.Add ("Images");
```

New CocosSharp templates include a class inheriting from `CCLayer`, implying that game visuals and logic should be added to this class. Instead, Fruity Falls uses multiple `CCLayer` instances to control draw order. These `CCLayer` instances are contained within the `GameView` class, which inherits from `CCScene`. Fruity Falls also includes multiple scenes, the first being the `TitleScene`. Therefore, the `Initialize` method instantiates a `TitleScene` instance which is passed to `RunWithScene`:

```
var scene = new TitleScene (GameView);
GameView.Director.RunWithScene (scene);
```

The content for Fruity Falls was created as low-resolution pixel art for aesthetic reasons. The `GameView.DesignResolution` is set so that the game is only 384 units wide and 512 tall:

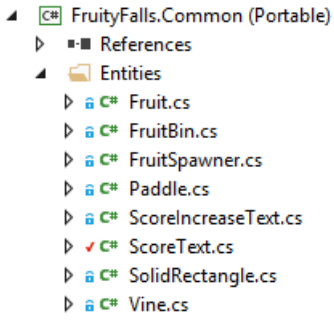
```
// We use a lower-resolution display to get a pixellated appearance
int width = 384;
int height = 512;
GameView.DesignResolution = new CCSizeI (width, height);
```

Finally, the `GameController` class provides a static method which can be called by any `CCGameScene` to transition to a different `CCScene`. This method is used to move between the `TitleScene` and the `GameScene`.

Game entities

Fruity Falls makes use of the entity pattern for most of the game objects. A detailed explanation of this pattern can be found in the Entities in CocosSharp guide.

The entity-implementing game objects can be found in the Entities folder in the **FruityFalls.Common** project:



Entities are objects which inherit from `CCNode`, and may have visuals, collision, and every-frame behavior.

The `Fruit` object represents a typical CocosSharp Entity: it contains a visual object, collision, and every-frame logic. Its constructor is responsible for initializing the Fruit:

```
public Fruit ()
{
 CreateFruitGraphic ();
 if (GameCoefficients.ShowCollisionAreas)
 {
  CreateDebugGraphic ();
 }
 CreateCollision ();
 CreateExtraPointsLabel ();
 Acceleration.Y = GameCoefficients.FruitGravity;
}
```

Fruit graphics

The `CreateFruitGraphic` method creates a `CCSprite` instance and adds it to the `Fruit`. The `IsAntialiased` property is set to false to give the game a pixelated look. This value is set to false on all `CCSprite` and `CCLabel` instances throughout the game:

```
private void CreateFruitGraphic()
{
 graphic = new CCSprite ("cherry.png");
    graphic.IsAntialiased = false;
 this.AddChild (graphic);
}
```

Whenever the player touches a `Fruit` instance with the `Paddle`, the point value of that fruit increases by one. This provides an extra challenge to experienced players to juggle fruit for extra points. The point value of the fruit is displayed using the `extraPointsLabel` instance.

The `CreateExtraPointsLabel` method creates a `CCLabel` instance and adds it to the `Fruit`:

```
private void CreateExtraPointsLabel()
{
 extraPointsLabel = new CCLabel("", "Arial", 12, CCLabelFormat.SystemFont);
 extraPointsLabel.IsAntialiased = false;
 extraPointsLabel.Color = CCColor3B.Black;
 this.AddChild(extraPointsLabel);
}
```

Fruity Falls includes two different types of fruit – cherries and lemons. The `CreateFruitGraphic` assigns a default visual, but the fruit visuals can be changed through the `FruitColor` property, which in turn calls `UpdateGraphics`:

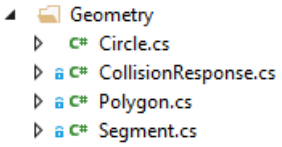
```
private void UpdateGraphics()
{
if (GameCoefficients.ShowCollisionAreas)
 {
  debugGrahic.Clear ();
  const float borderWidth = 4;
  debugGrahic.DrawSolidCircle (
   CCPoint.Zero,
   GameCoefficients.FruitRadius,
   CCColor4B.Black);
  debugGrahic.DrawSolidCircle (
   CCPoint.Zero,
   GameCoefficients.FruitRadius - borderWidth,
   fruitColor.ToCCColor ());
 }
 if (this.FruitColor == FruitColor.Yellow)
 {
  graphic.Texture = CCTextureCache.SharedTextureCache.AddImage ("lemon.png");
  extraPointsLabel.Color = CCColor3B.Black;
  extraPointsLabel.PositionY = 0;
 }
 else
 {
  graphic.Texture = CCTextureCache.SharedTextureCache.AddImage ("cherry.png");
  extraPointsLabel.Color = CCColor3B.White;
  extraPointsLabel.PositionY = -8;
 }
}
```

The first if-statement in `UpdateGraphics` updates the debug graphics, which are used to visualize collision areas. These visuals are turned off before a final release of the game is made, but can be kept on during development for debugging physics. The second part changes the `graphic.Texture` property by calling `CCTextureCache.SharedTextureCache.AddImage`. This method provides access to a texture by file name. More information on this method can be found in the Texture Caching guide.

The `extraPointsLabel` color is adjusted to keep contrast with the fruit image, and its `PositionY` value is adjusted to center the `CCLabel` on the fruit's `CCSprite`:

Collision

Fruity Falls implements a custom collision solution using objects in the Geometry folder:



Collision in Fruity Falls is implemented using either the `Circle` or `Polygon` object, usually with one of these two types being added as a child of an entity. For example, the `Fruit` object has a `Circle` called `Collision` which it initializes in its `CreateCollision` method:

```
private void CreateCollision()
{
 Collision = new Circle ();
 Collision.Radius = GameCoefficients.FruitRadius;
 this.AddChild (Collision);
}
```

Note that the `Circle` class inherits from the `CCNode` class, so it can be added as a child to our game's entities:

```
public class Circle : CCNode
{
 ...
}
```

`Polygon` creation is similar to `Circle` creation, as shown in the `Paddle` class:

```
private void CreateCollision()
{
    Polygon = Polygon.CreateRectangle(width, height);
 this.AddChild (Polygon);
}
```

Collision logic is covered later in this guide.

Physics

The physics in Fruity Falls can be separated into two categories: movement and collision.

Movement using velocity and acceleration

Fruity Falls uses `Velocity` and `Acceleration` values to control the movement of its entities, similar to the BouncingGame. Entities implement their movement logic in a method named `Activity`, which is called once per frame. For example, we can see the implementation of movement in the `Fruit` class' `Activity` method:

```
public void Activity(float frameTimeInSeconds)
{
    timeUntilExtraPointsCanBeAdded -= frameTimeInSeconds;

 // linear approximation:
 this.Velocity += Acceleration * frameTimeInSeconds;

    // This is a linear approximation to drag. It's used to
    // keep the object from falling too fast, and eventually
    // to slow its horizontal movement. This makes the game easier
    this.Velocity -= Velocity * GameCoefficients.FruitDrag * frameTimeInSeconds;

 this.Position += Velocity * frameTimeInSeconds;
}
```

The `Fruit` object is unique in its implementation of drag – a value which slows the velocity in relation to how fast the Fruit is moving. This implementation of drag provides a *terminal velocity*, which is the maximum speed that a Fruit instance can fall. Drag also slows down the horizontal movement of fruit, which makes the game slightly easier to play.

The `Paddle` object also applies `Velocity` in its `Activity` method, but its `Velocity` is calculated using a `desiredLocation` value:

```
public void Activity(float frameTimeInSeconds)
{
 // This code will cause the cursor to lag behind the touch point
 // Increasing this value reduces how far the paddle lags
    // behind the player's finger.
 const float velocityCoefficient = 4;
 // Get the velocity from current location and touch location
 Velocity = (desiredLocation - this.Position) * velocityCoefficient;
 this.Position += Velocity * frameTimeInSeconds;
 ...
}
```

Typically, games which use `Velocity` to control the position of their objects do not directly set entity positions, at least not after initialization. Rather than directly setting the `Paddle` position, the `Paddle.HandleInput` method assigns the `desiredLocation` value:

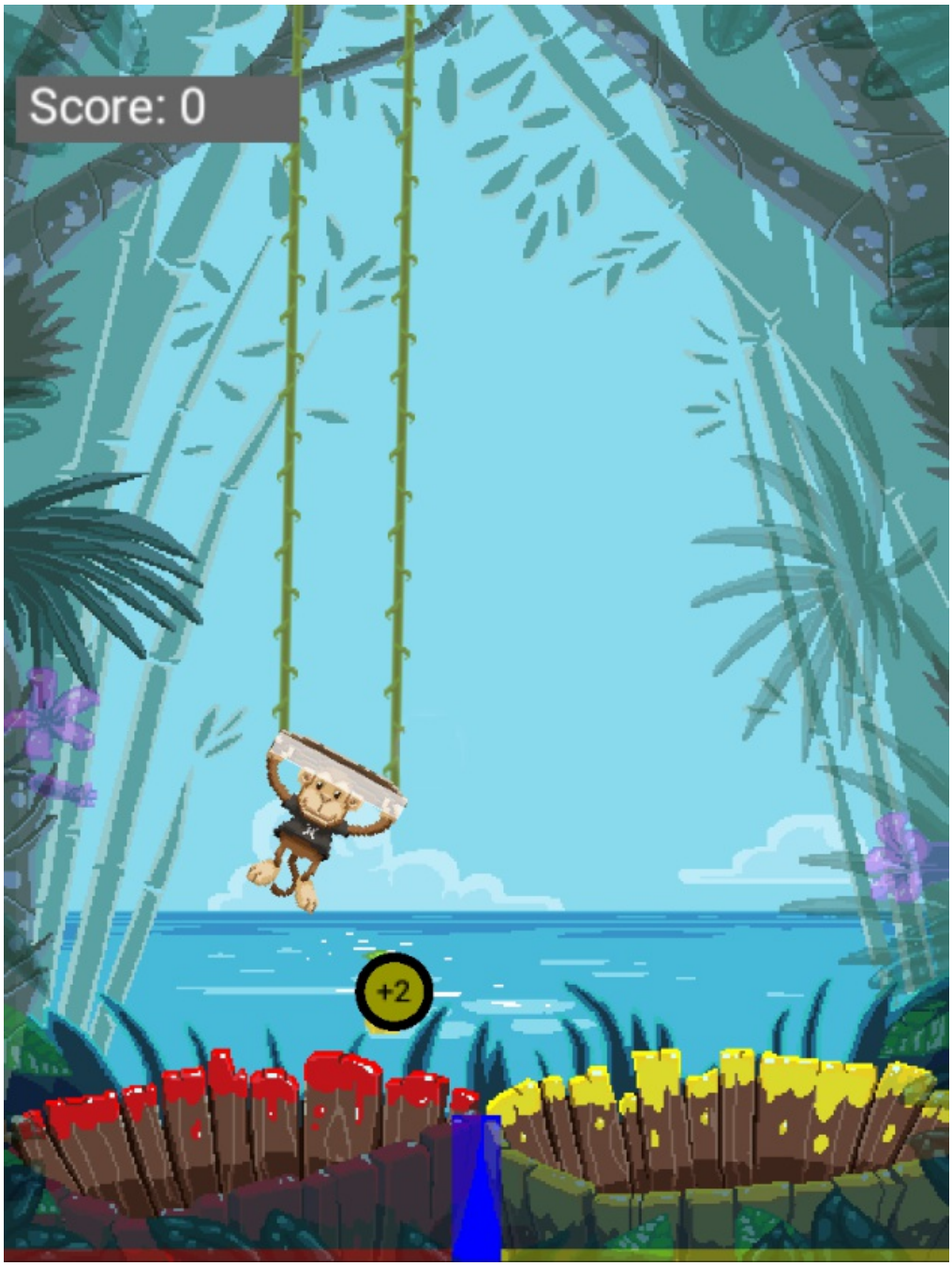
```
public void HandleInput(CCPoint touchPoint)
{
    desiredLocation = touchPoint;
}
```

Collision

Fruity Falls implements semi-realistic collision between the fruit and other collidable objects such as the `Paddle` and `GameScene.Splitter`. To help debug collision, Fruity Falls collision areas can be made visible by changing the `GameCoefficients.ShowDebugInfo` in the `GameCoefficients.cs` file:

```
public static class GameCoefficients
{
    ...
    public const bool ShowCollisionAreas = true;
 ...
}
```

Setting this value to `true` enables the rendering of collision areas:



Collision logic begins in the `GameScene.Activity` method:

```
private void Activity(float frameTimeInSeconds)
{
    if (hasGameEnded == false)
    {
        paddle.Activity(frameTimeInSeconds);
        foreach (var fruit in fruitList)
        {
            fruit.Activity(frameTimeInSeconds);
        }
        spawner.Activity(frameTimeInSeconds);
        DebugActivity();
        PerformCollision();
    }
}
```

`PerformCollision` handles colliding all `Fruit` instances with other objects:

```
private void PerformCollision()
{
    // reverse for loop since fruit may be destroyed:
    for(int i = fruitList.Count - 1; i > -1; i--)
    {
        var fruit = fruitList[i];
        FruitVsPaddle(fruit);
        FruitPolygonCollision(fruit, splitter.Polygon, CCPoint.Zero);
        FruitVsBorders(fruit);
        FruitVsBins(fruit);
    }
}
```

FruitVsBorders

`FruitVsBorders` collision performs its own logic for collision rather than relying on logic contained in a different class. This difference exists because the collision between fruit and the screen's borders is not perfectly solid – it is possible for fruit to be pushed off the edge of the screen by careful paddle movement. Fruit will bounce off of the screen when hit with the paddle, but if the player slowly pushes fruit it will move past the edge and off the screen:

```
private void FruitVsBorders(Fruit fruit)
{
    if(fruit.PositionX - fruit.Radius < 0 && fruit.Velocity.X < 0)
    {
        fruit.Velocity.X *= -1 * GameCoefficients.FruitCollisionElasticity;
    }
    if(fruit.PositionX + fruit.Radius > gameplayLayer.ContentSize.Width && fruit.Velocity.X > 0)
    {
        fruit.Velocity.X *= -1 * GameCoefficients.FruitCollisionElasticity;
    }
    if(fruit.PositionY + fruit.Radius > gameplayLayer.ContentSize.Height && fruit.Velocity.Y > 0)
    {
        fruit.Velocity.Y *= -1 * GameCoefficients.FruitCollisionElasticity;
    }
}
```

FruitVsBins

The `FruitVsBins` method is responsible for checking if any fruit has fallen into one of the two bins. If so, then the player is awarded points (if the fruit/bin colors match) or the game ends (if the colors do not match):

```
private void FruitVsBins(Fruit fruit)
{
    foreach(var bin in fruitBins)
    {
        if(bin.Polygon.CollideAgainst(fruit.Collision))
        {
            if(bin.FruitColor == fruit.FruitColor)
            {
                // award points:
                score += 1 + fruit.ExtraPointValue;
                scoreText.Score = score;
                Destroy(fruit);
            }
            else
            {
                EndGame();
            }
            break;
        }
    }
}
```

FruitVsPaddle and FruitPolygonCollision

Fruit vs. paddle and fruit vs. splitter (the area separating the two bins) collision both rely on the `FruitPolygonCollision` method. This method has three parts:

1. Test whether the objects collide
2. Move the objects (just the Fruit in Fruity Falls) so that they no longer overlap
3. Adjust the velocity of the objects (just the Fruit in Fruity Falls) to simulate a bounce The following code demonstrates the points made above:

```
private static bool FruitPolygonCollision(Fruit fruit, Polygon polygon, CCPoint polygonVelocity)
{
    // Test whether the fruit collides
    bool didCollide = polygon.CollideAgainst(fruit.Collision);
    if (didCollide)
    {
        var circle = fruit.Collision;
        // Get the separation vector to reposition the fruit so it doesn't overlap the polygon
        var separation = CollisionResponse.GetSeparationVector(circle, polygon);
        fruit.Position += separation;
        // Adjust the fruit's Velocity to make it bounce:
        var normal = separation;
        normal.Normalize();
        fruit.Velocity = CollisionResponse.ApplyBounce(
            fruit.Velocity,
            polygonVelocity,
            normal,
            GameCoefficients.FruitCollisionElasticity);
    }
    return didCollide;
}
```

Fruity Falls collision response is one-sided – only the fruit's velocity and position are adjusted. It's worth noting that other games may have collision which impacts both objects involved, such as a character pushing a boulder or two cars crashing into each other.

The math behind the collision logic contained in the `Polygon` and `CollisionResponse` classes is beyond the scope of this guide, but as-written, these methods can be used for many types of games. The Polygon and `CollisionResponse` classes even support non-rectangular and convex polygons, so this code can be used for many types of games.

Game content

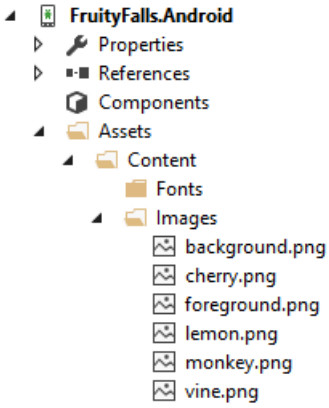
The art in Fruity Falls immediately distinguishes the game from the BouncingGame. While the game designs are similar, players will immediately see a difference in how the two games look. Gamers often decide whether to try a game by its visuals. Therefore, it is vitally important that developers invest resources in making a visually appealing game.

The art for Fruity Falls was created with the following goals:

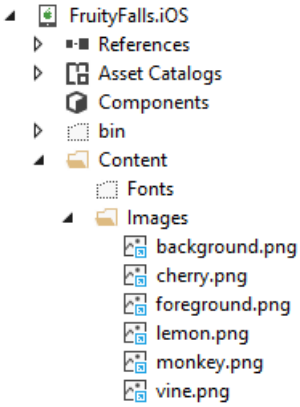
- Consistent theme
- Emphasis on just one character – the Xamarin monkey
- Bright colors to create a relaxing, enjoyable experience
- Contrast between the background and foreground so gameplay objects are easy to track visually
- Ability to create simple visual effects without resource-heavy animations

Content location

Fruity Falls includes all of its content in the Images folder in the Android project:



These same files are linked in the iOS project to avoid duplication, and so changes to the files impact both projects:



It's worth noting that the content is not contained within the **Ld** or **Hd** folders, which are part of the default CocosSharp template. The **Ld** and **Hd** folders are intended to be used for games which provide two sets of content – one for lower-resolution devices, such as phones, and one for higher-resolution devices, such as tablets. The Fruity Falls art is intentionally created with a pixelated aesthetic, so it does not need to provide content for different screen sizes. Therefore, the **Ld** and **Hd** folders have been completely removed from the project.

GameScene layering

As mentioned earlier in this guide, the `GameScene` is responsible for all game object instantiation, positioning, and inter-object logic (such as collision). All objects are added to one of four `CCLayer` instances:

```
CCLayer backgroundLayer;
CCLayer gameplayLayer;
CCLayer foregroundLayer;
CCLayer hudLayer;
```

These four layers are created in the `CreateLayers` method. Note that they are added to the `GameScene` in order of back-to-front:

```
private void CreateLayers()
{
    backgroundLayer = new CCLayer();
    this.AddLayer(backgroundLayer);
    gameplayLayer = new CCLayer();
    this.AddLayer(gameplayLayer);
    foregroundLayer = new CCLayer();
    this.AddLayer(foregroundLayer);
    hudLayer = new CCLayer();
    this.AddLayer(hudLayer);
}
```

Once the layers are created, all visual objects are added to the layers using the `AddChild` method, as shown in the

`CreateBackground` method:

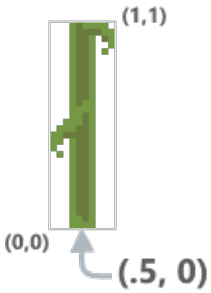
```
private void CreateBackground()
{
    var background = new CCSprite("background.png");
    background.AnchorPoint = new CCPoint(0, 0);
    background.IsAntialiased = false;
    backgroundLayer.AddChild(background);
}
```

Vine entity

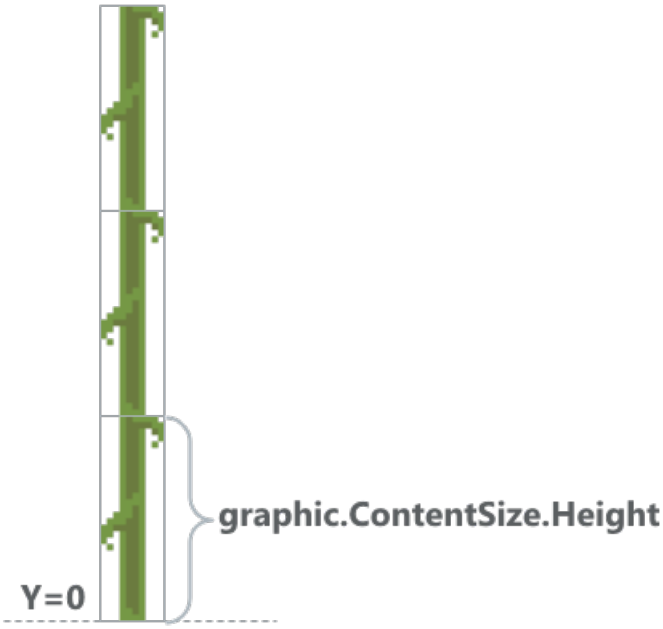
The `Vine` entity is uniquely used for content – it has no impact on gameplay. It is composed of twenty `CCSprite` instances, a number selected by trial and error to make sure the vine always reaches the top of the screen:

```
public Vine ()
{
 const int numberOfVinesToAdd = 20;
 for (int i = 0; i < numberOfVinesToAdd; i++)
 {
  var graphic = new CCSprite ("vine.png");
        graphic.AnchorPoint = new CCPoint(.5f, 0);
  graphic.PositionY = i * graphic.ContentSize.Height;
  this.AddChild (graphic);
 }
}
```

The first `CCSprite` is positioned so its bottom edge matches the position of the vine. This is accomplished by setting the AnchorPoint to `new CCPoint(.5f, 0)`. The `AnchorPoint` property uses *normalized coordinates* which range between 0 and 1 regardless of the size of the `CCSprite`:



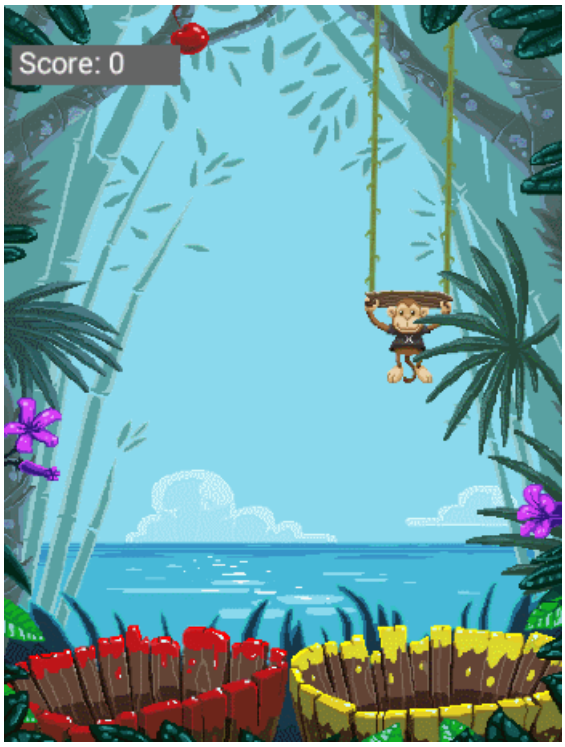
Subsequent vine sprites are positioned using the `graphic.ContentSize.Height` value, which returns the height of the image in pixels. The following diagram shows how the vine graphic tiles upward:



Since the origin of the `Vine` entity is at the bottom of the vine, then positioning it is straightforward, as is shown in the `Paddle.CreateVines` method:

```
private void CreateVines()
{
    // Increasing this value moves the vines closer to the
    // center of the paddle.
    const int vineDistanceFromEdge = 4;
    leftVine = new Vine ();
    var leftEdge = -width / 2.0f;
    leftVine.PositionX = leftEdge + vineDistanceFromEdge;
    this.AddChild (leftVine);
    rightVine = new Vine ();
    var rightEdge = width / 2.0f;
    rightVine.PositionX = rightEdge - vineDistanceFromEdge;
    this.AddChild (rightVine);
}
```

The Vine instances in the `Paddle` entity also have interesting rotation behavior. Since the `Paddle` entity as a whole rotates according to player input (which this guide covers in more detail below), the `Vine` instances are also impacted by this rotation. However, rotating the `Vine` instances along with the `Paddle` produces an undesirable visual effect as shown in the following animation:



To counteract the `Paddle` rotation, the `leftVine` and `rightVine` instances are rotated the opposite direction of the paddle, as shown in the `Paddle.Activity` method:

```
public void Activity(float frameTimeInSeconds)
{
 ...
 // This value adds a slight amount of rotation
    // to the vine. Having a small amount of tilt looks nice.
 float vineAngle = this.Velocity.X / 100.0f;
 leftVine.Rotation = -this.Rotation + vineAngle;
 rightVine.Rotation = -this.Rotation + vineAngle;
}
```

Notice that a small amount of rotation is added back to the vine through the `vineAngle` coefficient. This value can be changed to adjust how much the vines rotate.

GameCoefficients

Every good game is the product of iteration, so Fruity Falls includes a class called `GameCoefficients` which controls how the game is played. This class contains expressive variables which are used throughout the game to control physics, layout, spawning, and scoring.

For example, the Fruit physics are controlled by the following variables:

```
public static class GameCoefficients
{
    ...

    // The strength of the gravity. Making this a
    // smaller (bigger negative) number will make the
    // fruit fall faster. A larger (smaller negative) number
    // will make the fruit more floaty.
    public const float FruitGravity = -60;
    // The impact of "air drag" on the fruit, which gives
    // the fruit terminal velocity (max falling speed) and slows
    // the fruit's horizontal movement (makes the game easier)
    public const float FruitDrag = .1f;
    // Controls fruit collision bouncyness. A value of 1
    // means no momentum is lost. A value of 0 means all
    // momentum is lost. Values greater than 1 create a spring-like effect
    public const float FruitCollisionElasticity = .5f;

    ...
}
```

As the names imply, these variables can be used to adjust how fast fruit falls, how horizontal movement slows down over time, and paddle bounciness.

Game coefficients stored in code or data files (such as XML) can be especially valuable for teams with game designers who are not also programmers.

The `GameCoefficients` class also includes values for turning on debug information such as spawning information or collision areas:

```
public static class GameCoefficients
{
    ...
    // This controls whether debug information is displayed on screen.
    public const bool ShowDebugInfo = false;
    public const bool ShowCollisionAreas = false;
    ...
}
```

Conclusion

This guide explored the Fruity Falls game. It covered concepts including content, physics, and game state management.

Related links

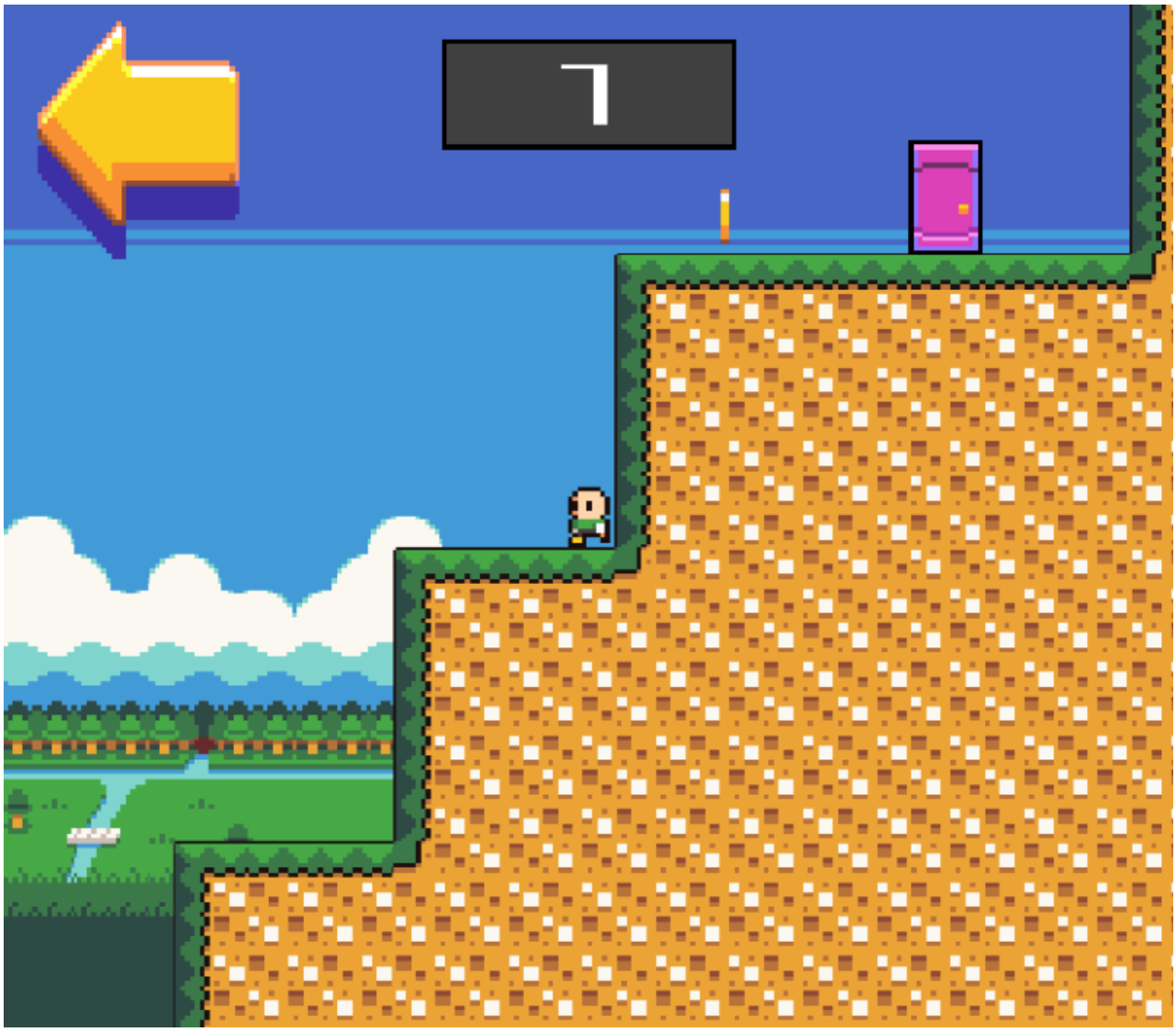
- CocosSharp API Documentation
- Completed project (sample)

Coin Time game details

10/3/2018 • 14 minutes to read • Edit Online

This guide discusses implementation details in the Coin Time game, including working with tile maps, creating entities, animating sprites, and implementing efficient collision.

Coin Time is a full platformer game for iOS and Android. The goal of the game is to collect all of the coins in a level and then reach the exit door while avoiding enemies and obstacles.



This guide discusses implementation details in Coin Time, covering the following topics:

- Working with tmx files
- Level loading
- Adding new entities
- Animated entities

Content in Coin Time

Coin Time is a sample project that represents how a full CocosSharp project might be organized. Coin Time's structure aims to simplify the addition and maintenance of content. It uses **.tmx** files created by Tiled for levels and XML files to define animations. Modifying or adding new content can be achieved with minimal effort.

While this approach makes Coin Time an effective project for learning and experimentation, it also reflects how professional games are made. This guide explains some of the approaches taken to simplify adding and modifying content.

Working with tmx files

Coin Time levels are defined using the .tmx file format, which is output by the Tiled tile map editor. For a detailed discussion of working with Tiled, see the Using Tiled with Cocos Sharp guide.

Each level is defined in its own .tmx file contained in the **CoinTime/Assets/Content/levels** folder. All Coin Time levels share one tileset file, which is defined in the **mastersheet.tsx** file. This file defines the custom properties for each tile, such as whether the tile has solid collision or whether the tile should be replaced by an entity instance. The mastersheet.tsx file allows properties to be defined only once and used across all levels.

Editing a tile map

To edit a tile map, open the .tmx file in Tiled by double-clicking the .tmx file or opening it through the File menu in Tiled. Coin Time level tile maps contain three layers:

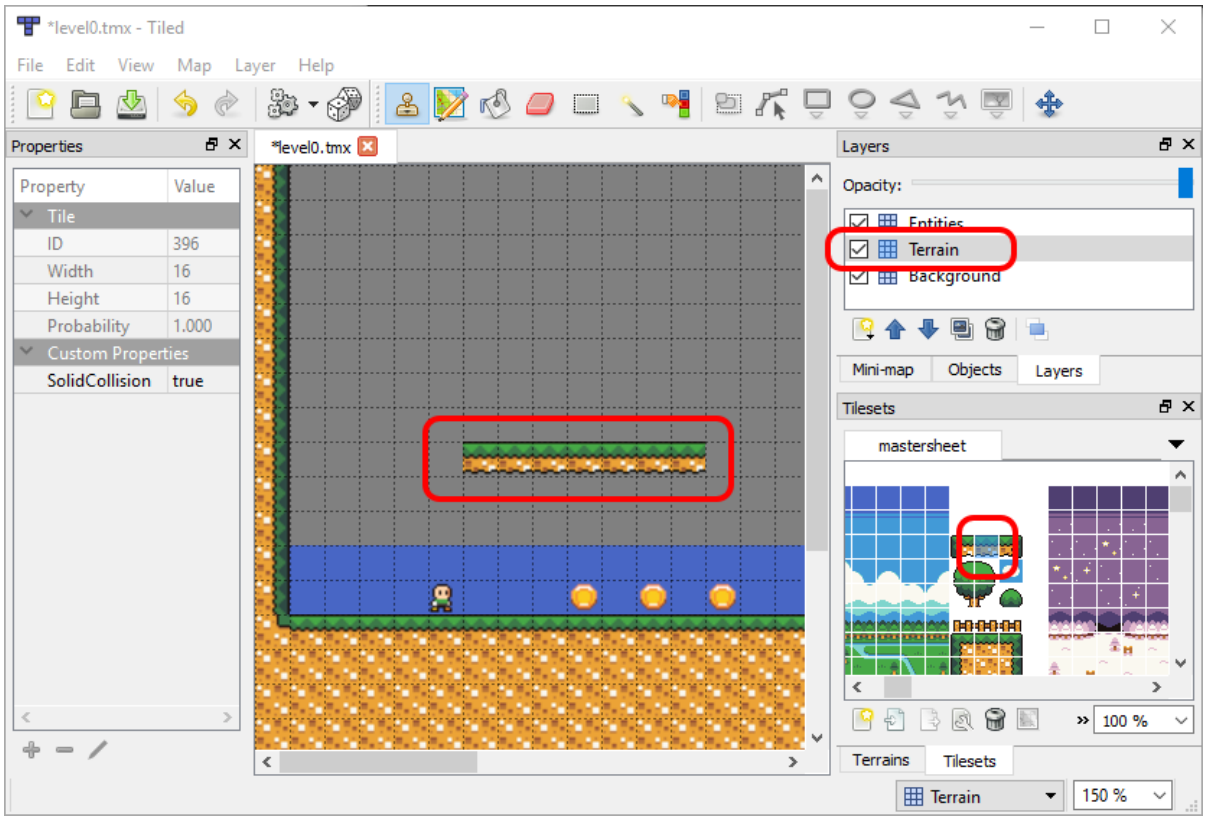
- **Entities** – this layer contains tiles which will be replaced with instances of entities at runtime. Examples include the player, coins, enemies, and the end-of-level door.
- **Terrain** – this layer contains tiles which typically have solid collision. Solid collision allows the player to walk on these tiles without falling through. Tiles with solid collision can also act as walls and ceilings.
- **Background** – the Background layer contains tiles that are used as the static background. This layer does not scroll when the camera moves throughout the level, creating the appearance of depth through parallax.

As we will explore later, the level-loading code expects these three layers in all Coin Time levels.

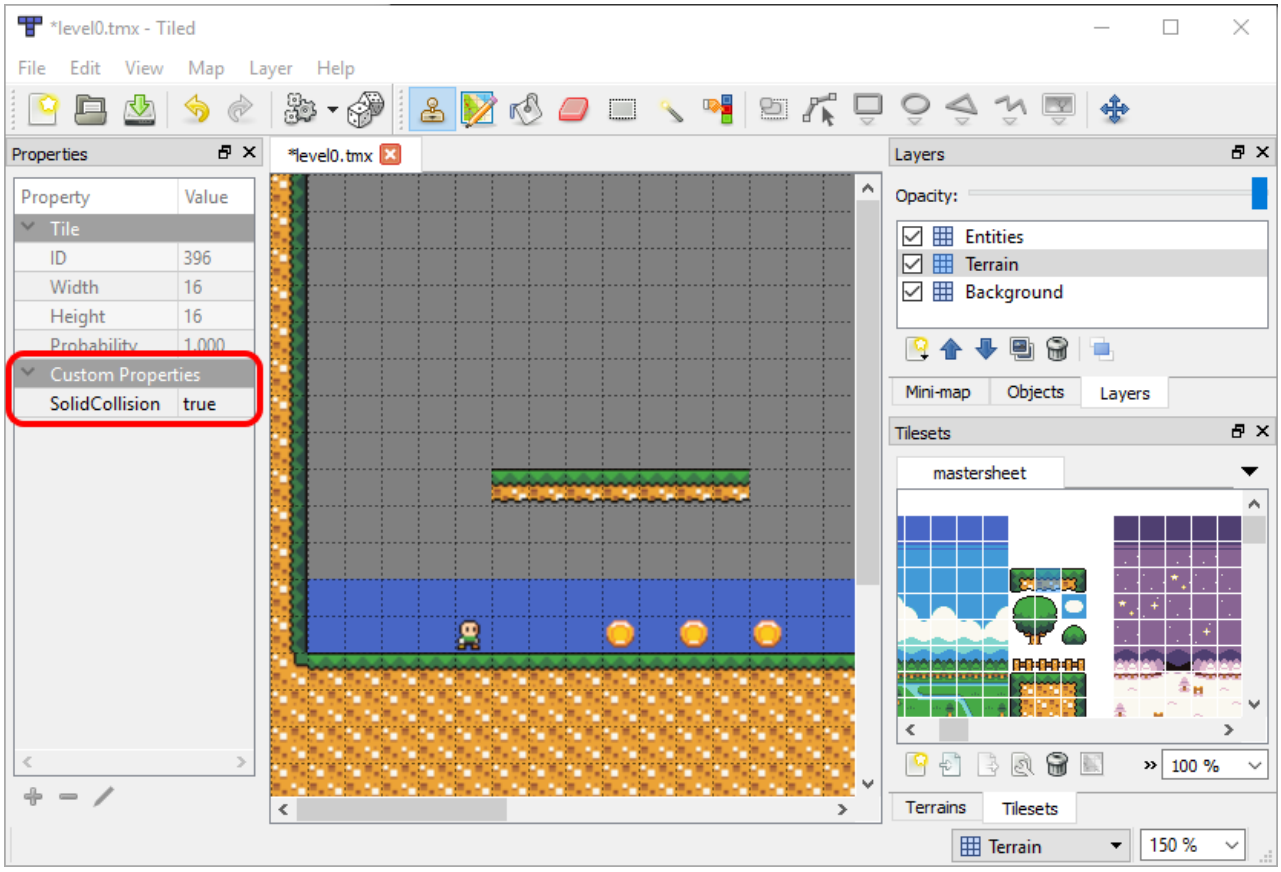
Editing terrain

Tiles can be placed by clicking in the **mastersheet** tileset and then clicking on the tile map. For example, to paint new terrain in a level:

1. Select the Terrain layer
2. Click on the tile to draw
3. Click or push and drag over the map to paint the tile

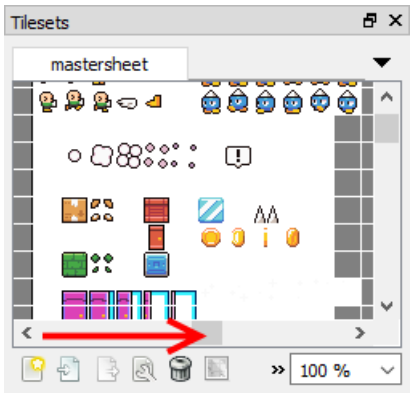


The top-left of the tileset contains all of the terrain in Coin Time. Terrain, which is solid, includes the **SolidCollision** property, as shown in the tile properties on the left of the screen:

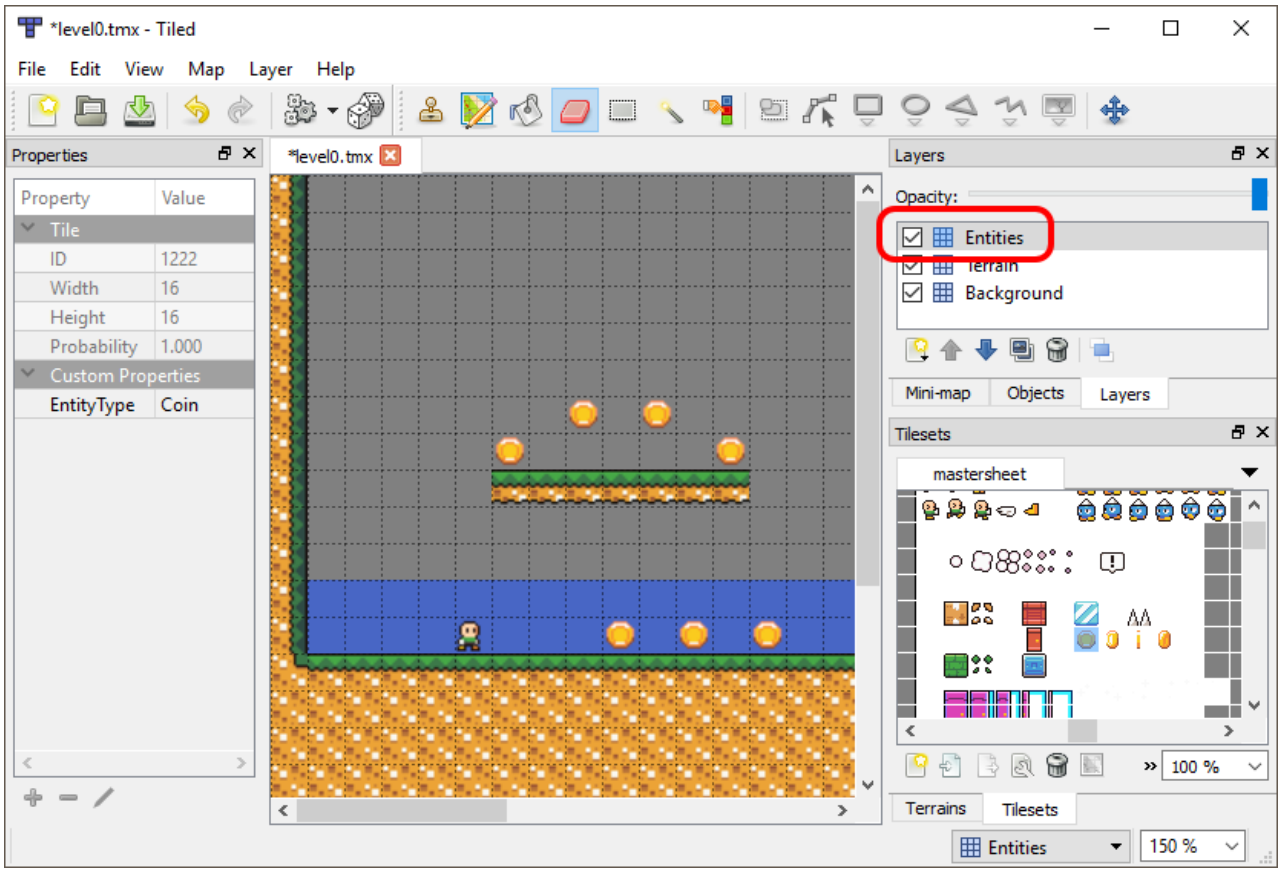


Editing entities

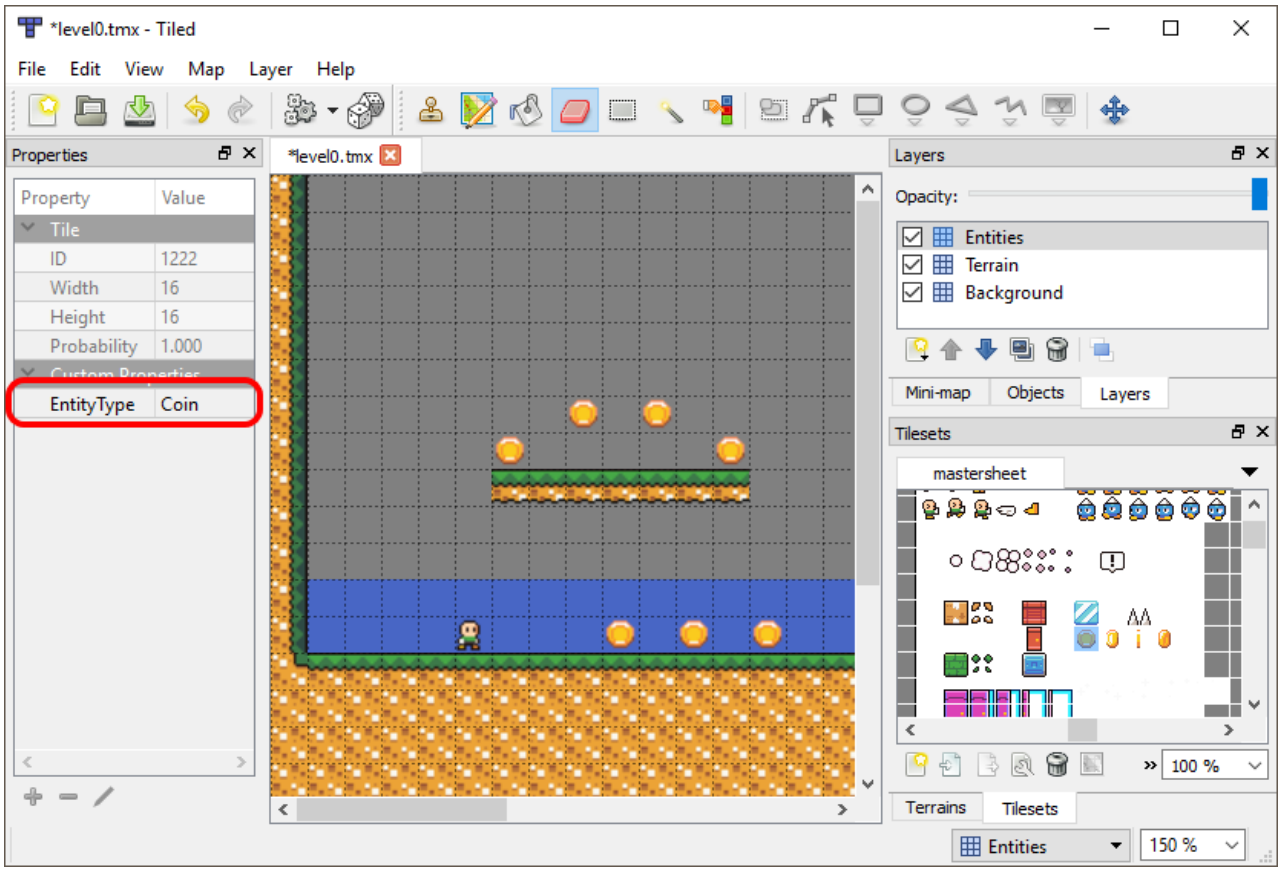
Entities can be added or removed from a level – just like terrain. The **mastersheet** tileset has all entities placed about halfway horizontally, so they may not be visible without scrolling to the right:



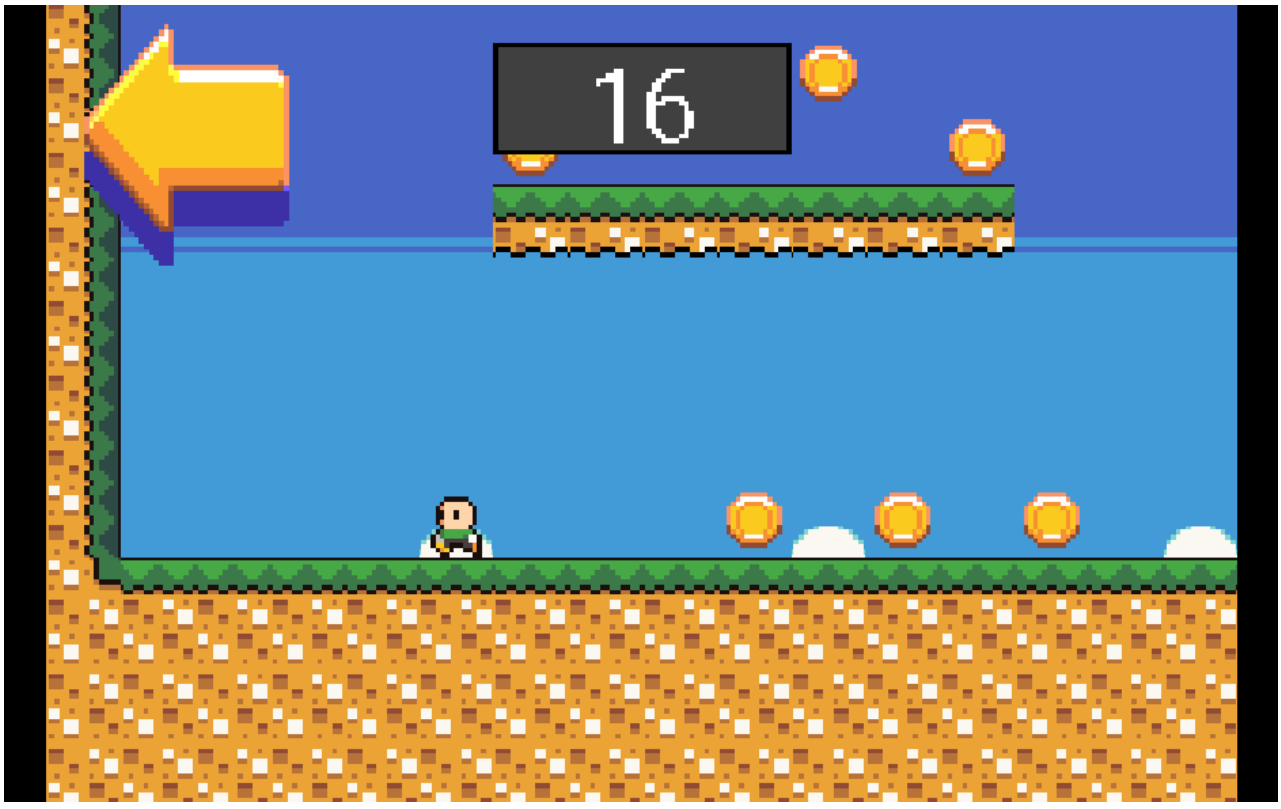
New entities should be placed on the **Entities** layer.



CoinTime code looks for the **EntityType** when a level is loaded to identify tiles which should be replaced by entities:



Once the file has been modified and saved, the changes will automatically show up if the project is built and run:



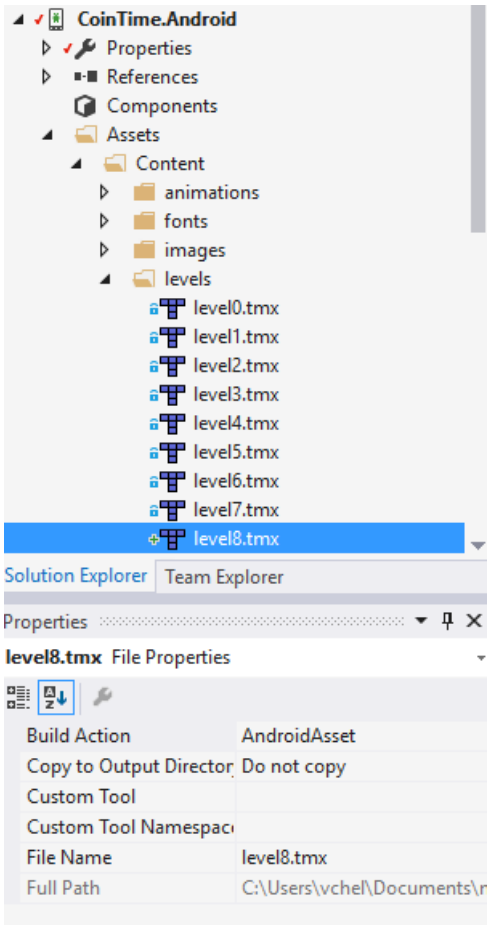
Adding new levels

The process of adding levels to Coin Time requires no code changes, and only a few small changes to the project. To add a new level:

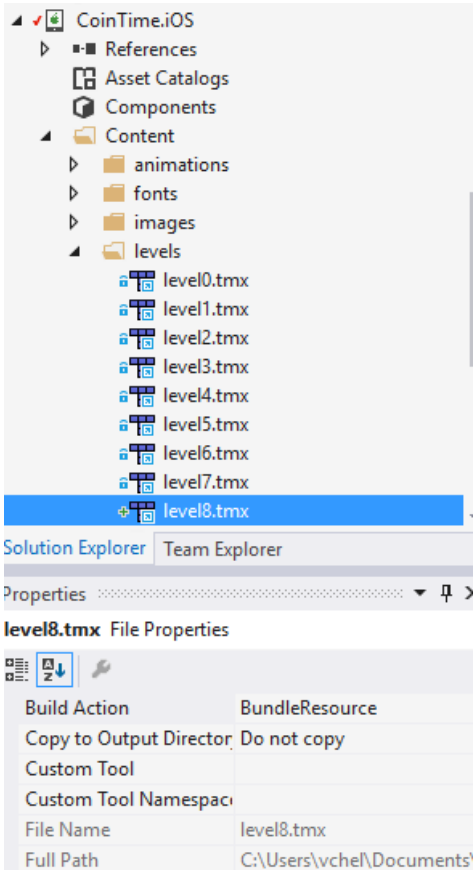
1. Open the level folder located at <**CoinTime Root**>**\CoinTime\Assets\Content\levels**
2. Copy and paste one of the levels, such as **level0.tmx**

3. Rename the new .tmx file so it continues the level number sequence with existing levels, such as **level8.tmx**

4. In Visual Studio or Visual Studio for Mac, add the new .tmx file to the Android levels folder. Verify that the file uses the **AndroidAsset** build action.



5. Add the new .tmx file to the iOS levels folder. Be sure to link the file from its original location and verify that it uses the **BundleResource** build action.



The new level should appear in the level select screen as level 9 (level file names start at 0, but the level buttons begin with the number 1):

Level loading

As shown earlier, new levels require no changes in code – the game automatically detects the levels if they are named correctly and added to the **levels** folder with the correct build action (**BundleResource** or **AndroidAsset**).

The logic for determining the number of levels is contained in the `LevelManager` class. When an instance of the `LevelManager` is constructed (using the singleton pattern), the `DetermineAvailbleLevels` method is called:

```
private void DetermineAvailableLevels()
{
 // This game relies on levels being named "levelx.tmx" where x is an integer beginning with
 // 1. We have to rely on MonoGame's TitleContainer which doesn't give us a GetFiles method - we simply
 // have to check if a file exists, and if we get an exception on the call then we know the file doesn't
 // exist.
 NumberOfLevels = 0;
 while (true)
 {
  bool fileExists = false;
  try
  {
   using(var stream = TitleContainer.OpenStream("Content/levels/level" + NumberOfLevels + ".tmx"))
   {
   }
   // if we got here then the file exists!
   fileExists = true;
  }
  catch
  {
   // do nothing, fileExists will remain false
  }
  if (!fileExists)
  {
   break;
  }
  else
  {
   NumberOfLevels++;
  }
 }
}
```

CocosSharp does not provide a cross-platform approach for detecting if files are present, so we have to rely on the `TitleContainer` class to attempt to open a stream. If the code for opening a stream throws an exception, then the file does not exist and the while loop breaks. Once the loop finishes, the `NumberOfLevels` property reports how many valid levels are part of the project.

The `LevelSelectScene` class uses the `LevelManager.NumberOfLevels` to determine how many buttons to create in the `CreateLevelButtons` method:

```
private void CreateLevelButtons()
{
 const int buttonsPerPage = 6;
 int levelIndex0Based = buttonsPerPage * pageNumber;
 int maxLevelExclusive = System.Math.Min (levelIndex0Based + 6, LevelManager.Self.NumberOfLevels);
 int buttonIndex = 0;
 float centerX = this.ContentSize.Center.X;
 const float topRowOffsetFromCenter = 16;
 float topRowY = this.ContentSize.Center.Y + topRowOffsetFromCenter;
 for (int i = levelIndex0Based; i < maxLevelExclusive; i++)
 {
  ...
 }
}
```

The `NumberOflevels` property is used to determine which buttons should be created. This code considers which page the user is currently viewing and only creates a maximum of six buttons per page. When clicked, the button instances call the `HandleButtonClicked` method:

```
private void HandleButtonClicked(object sender, EventArgs args)
{
 // levelNumber is 1-based, so subtract 1:
 var levelIndex = (sender as Button).LevelNumber - 1;
 LevelManager.Self.CurrentLevel = levelIndex;
 CoinTime.GameAppDelegate.GoToGameScene ();
}
```

This method assigns the `CurrentLevel` property which is used by the `GameScene` when loading a level. After setting the `CurrentLevel`, the `GoToGameScene` method is raised, switching the scene from `LevelSelectScene` to `GameScene`.

The `GameScene` constructor calls `GoToLevel`, which performs the level-loading logic:

```
private void GoToLevel(int levelNumber)
{
 LoadLevel (levelNumber);
 CreateCollision();
 ProcessTileProperties ();
 touchScreen = new TouchScreenInput(gameplayLayer);
 secondsLeft = secondsPerLevel;
}
```

Next we'll take a look at methods called in `GoToLevel`.

LoadLevel

The `LoadLevel` method is responsible for loading the .tmx file and adding it to the `GameScene`. This method does not create any interactive objects such as collision or entities – it simply creates the visuals for the level, also referred to as the *environment*.

```
private void LoadLevel(int levelNumber)
{
 currentLevel = new CCTileMap ("level" + levelNumber + ".tmx");
 currentLevel.Antialiased = false;
 backgroundLayer = currentLevel.LayerNamed ("Background");
 // CCTileMap is a CCLayer, so we'll just add it under all entities
 this.AddChild (currentLevel);
 // put the game layer after
 this.RemoveChild(gameplayLayer);
 this.AddChild(gameplayLayer);
 this.RemoveChild (hudLayer);
 this.AddChild (hudLayer);
}
```

The `CCTileMap` constructor takes a file name, which is created using the level number passed in to `LoadLevel`. For more information on creating and working with `CCTileMap` instances, see the Using Tiled with CocosSharp guide.

Currently, CocosSharp does not allow reordering of layers without removing and re-adding them to their parent `CCScene` (which is the `GameScene` in this case), so the last few lines of the method are required to reorder the layers.

CreateCollision

The `CreateCollision` method constructs a `LevelCollision` instance which is used to perform *solid collision* between the player and environment.

```
private void CreateCollision()
{
    levelCollision = new LevelCollision();
    levelCollision.PopulateFrom(currentLevel);
}
```

Without this collision, the player would fall through the level and the game would be unplayable. Solid collision lets the player walk on the ground and prevents the player from walking through walls or jumping up through ceilings.

Collision in Coin Time can be added with no additional code – only modifications to tiled files.

ProcessTileProperties

Once a level is loaded and the collision is created, `ProcessTileProperties` is called to perform logic based on tile properties. Coin Time includes a `PropertyLocation` struct for defining properties and the coordinates of the tile with these properties:

```
public struct PropertyLocation
{
 public CCTileMapLayer Layer;
 public CCTileMapCoordinates TileCoordinates;
 public float WorldX;
 public float WorldY;
 public Dictionary<string, string> Properties;
}
```

This struct is used to construct create entity instances and remove unnecessary tiles in the `ProcessTileProperties` method:

```
private void ProcessTileProperties()
{
 TileMapPropertyFinder finder = new TileMapPropertyFinder (currentLevel);
 foreach (var propertyLocation in finder.GetPropertyLocations())
 {
  var properties = propertyLocation.Properties;
  if (properties.ContainsKey ("EntityType"))
  {
   float worldX = propertyLocation.WorldX;
   float worldY = propertyLocation.WorldY;
   if (properties.ContainsKey ("YOffset"))
   {
    string yOffsetAsString = properties ["YOffset"];
    float yOffset = 0;
    float.TryParse (yOffsetAsString, out yOffset);
    worldY += yOffset;
   }
   bool created = TryCreateEntity (properties ["EntityType"], worldX, worldY);
   if (created)
   {
    propertyLocation.Layer.RemoveTile (propertyLocation.TileCoordinates);
   }
  }
  else if (properties.ContainsKey ("RemoveMe"))
  {
   propertyLocation.Layer.RemoveTile (propertyLocation.TileCoordinates);
  }
 }
}
```

The foreach loop evaluates each tile property, checking if the key is either `EntityType` or `RemoveMe`. `EntityType`

indicates that an entity instance should be created. `RemoveMe` indicates that the tile should be completely removed at runtime.

If a property with the `EntityType` key is found, then `TryCreateEntity` is called, which attempts to create an entity using the property matching the `EntityType` key:

```
private bool TryCreateEntity(string entityType, float worldX, float worldY)
{
 CCNode entityAsNode = null;
 switch (entityType)
 {
 case "Player":
  player = new Player ();
  entityAsNode = player;
  break;
 case "Coin":
  Coin coin = new Coin ();
  entityAsNode = coin;
  coins.Add (coin);
  break;
 case "Door":
  door = new Door ();
  entityAsNode = door;
  break;
 case "Spikes":
  var spikes = new Spikes ();
  this.damageDealers.Add (spikes);
  entityAsNode = spikes;
  break;
 case "Enemy":
  var enemy = new Enemy ();
  this.damageDealers.Add (enemy);
  this.enemies.Add (enemy);
  entityAsNode = enemy;
  break;
 }
 if(entityAsNode != null)
 {
  entityAsNode.PositionX = worldX;
  entityAsNode.PositionY = worldY;
  gameplayLayer.AddChild (entityAsNode);
 }
 return entityAsNode != null;
}
```

Adding new entities

Coin Time uses the entity pattern for its game objects (which is covered in the Entities in CocosSharp guide). All entities inherit from `CCNode`, which means they can be added as children of the `gameplayLayer`.

Each entity type is also referenced directly through a list or single instance. For example, the `Player` is referenced by the `player` field, and all `Coin` instances are referenced in a `coins` list. Keeping direct references to entities (as opposed to referencing them through the `gameLayer.Children` list) makes code which accesses these entities easier to read and eliminates potentially expensive type casting.

The existing code provides a number of entity types as examples of how to create new entities. The following steps can be used to create a new entity:

1 - Define a new class using the entity pattern

The only requirement for creating an entity is to create a class which inherits from `CCNode`. Most entities have some visual, such as a `CCSprite`, which should be added as a child of the entity in its constructor.

CoinTime provides the `AnimatedSpriteEntity` class which simplifies the creation of animated entities. Animations will be covered in more detail in the Animated Entities section.

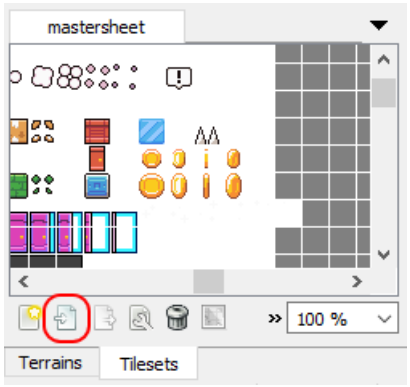
2 – Add a new entry to the TryCreateEntity switch statement

Instances of the new entity should be instantiated in the `TryCreateEntity`. If the entity requires every-frame logic like collision, AI, or reading input, then the `GameScene` needs to keep a reference to the object. If multiple instances are needed (such as `Coin` or `Enemy` instances), then a new `List` should be added to the `GameScene` class.

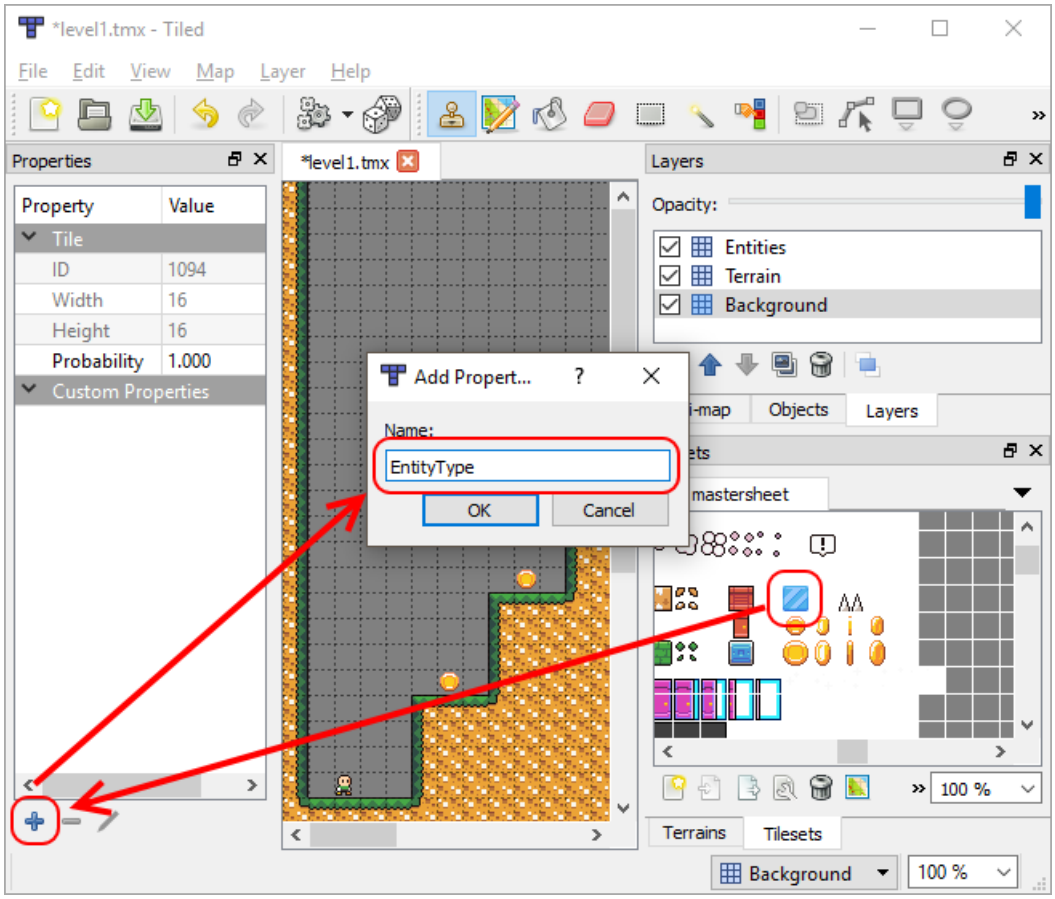
3 – Modify tile properties for the new entity

Once the code supports the creation of the new entity, the new entity needs to be added to the tileset. The tileset can be edited by opening any level `.tmx` file.

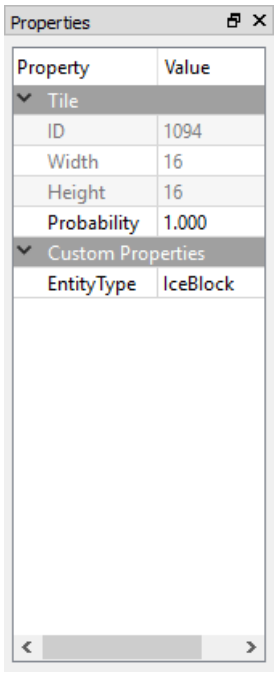
The tileset is stored separate from the .tmx in the **mastersheet.tsx** file, so it must be imported before it can be edited:



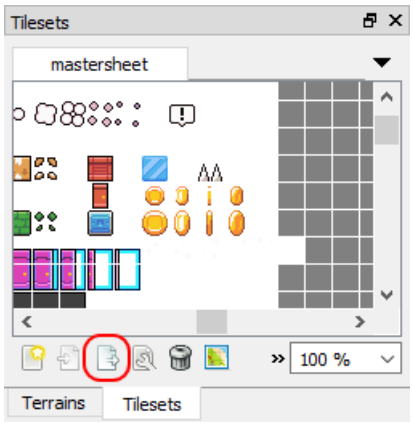
Once imported, properties on selected tiles are editable, and the EntityType can be added:



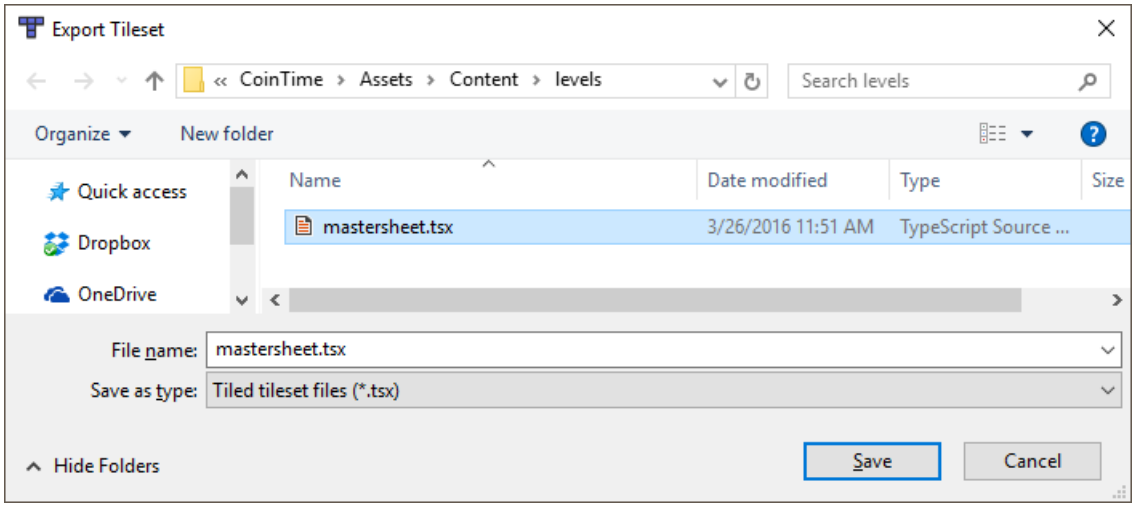
After the property is created, its value can be set to match the new `case` in `TryCreateEntity`:



After the tileset has been changed, it must be exported – this makes the changes available for all other levels:



The tileset should overwrite the existing **mastersheet.tsx** tileset:



Entity tile removal

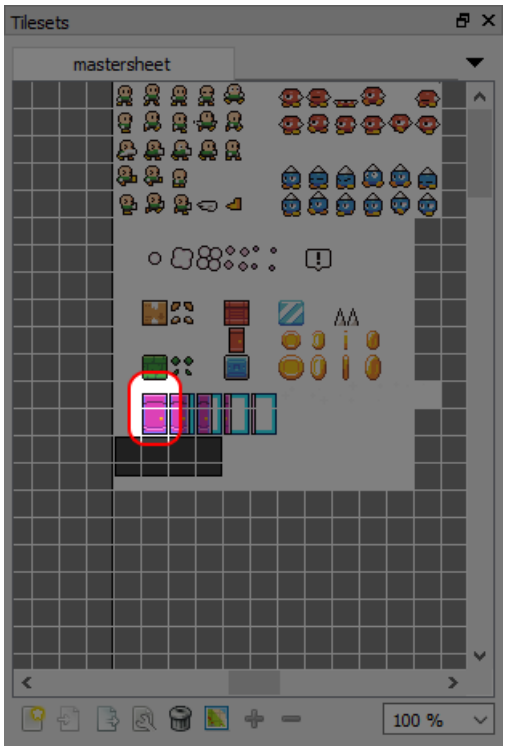
When a tile map is loaded into a game, the individual tiles are static objects. Since entities require custom behavior such as movement, Coin Time code removes tiles when entities are created.

`ProcessTileProperties` includes logic to remove tiles which create entities using the `RemoveTile` method:

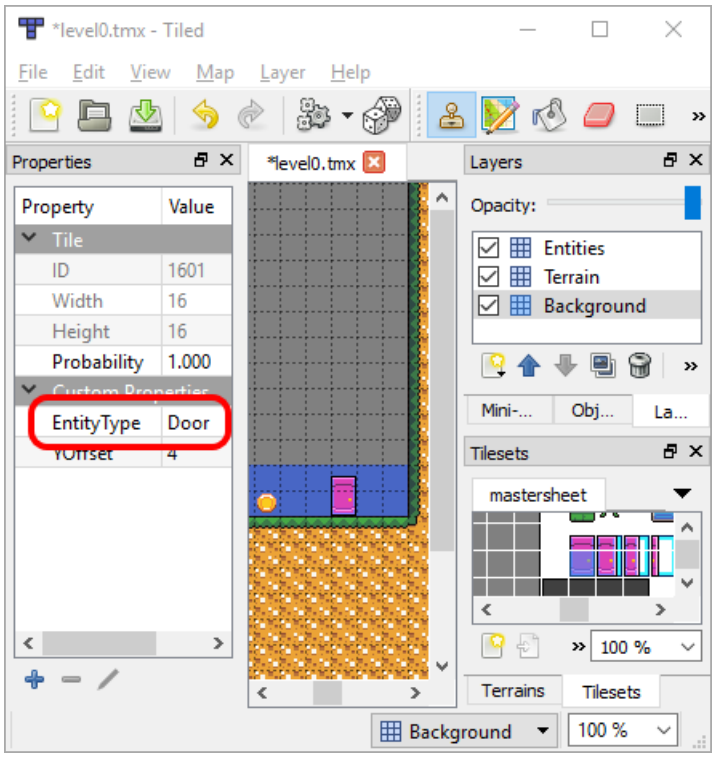
```
private void ProcessTileProperties()
{
 TileMapPropertyFinder finder = new TileMapPropertyFinder (currentLevel);
 foreach (var propertyLocation in finder.GetPropertyLocations())
 {
  var properties = propertyLocation.Properties;
  if (properties.ContainsKey ("EntityType"))
  {
   ...
   bool created = TryCreateEntity (properties ["EntityType"], worldX, worldY);
   if (created)
   {
    propertyLocation.Layer.RemoveTile (propertyLocation.TileCoordinates);
   }
  }
  ...
 }
}
```

This automatic removal of tiles is sufficient for entities which occupy only one tile in the tileset, such as coins and enemies. Larger entities require additional logic and properties.

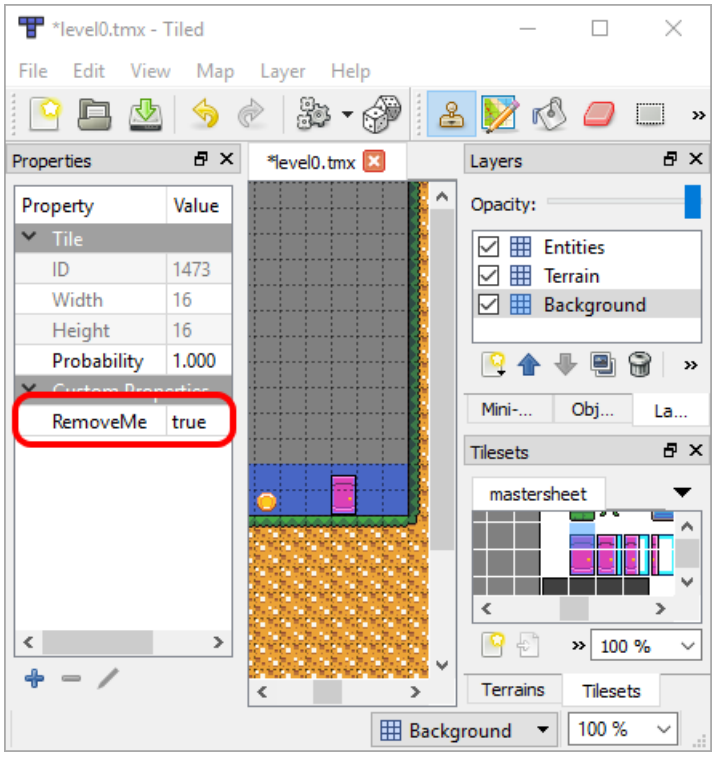
The Door requires two tiles to be drawn completely:



The bottom tile in the door contains the properties for creating an entity (**EntityType** set to **Door**):



Since only the bottom tile in the door is removed when the Door instance is created, additional logic is needed to remove the top tile at runtime. The top tile has a **RemoveMe** property set to **true**:



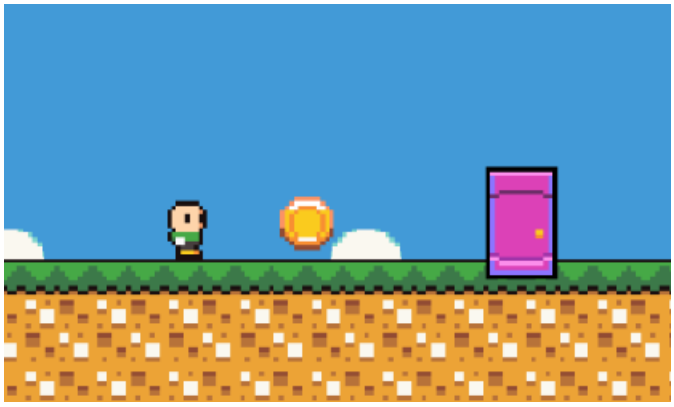
This property is used to remove tiles in `ProcessTileProperties`:

```
private void ProcessTileProperties()
{
 TileMapPropertyFinder finder = new TileMapPropertyFinder (currentLevel);
 foreach (var propertyLocation in finder.GetPropertyLocations())
 {
  var properties = propertyLocation.Properties;
  ...
  else if (properties.ContainsKey ("RemoveMe"))
  {
   propertyLocation.Layer.RemoveTile (propertyLocation.TileCoordinates);
  }
 }
}
```

Entity offsets

Entities created from tiles are positioned by aligning the center of the entity with the center of the tile. Larger entities, like `Door`, use additional properties and logic to be placed correctly.

The bottom door tile, which defines the `Door` entity placement, specifies a **YOffset** value of 4. Without this property, the `Door` instance is placed at the center of the tile:



This is corrected by applying the **YOffset** value in `ProcessTileProperties`:

```
private void ProcessTileProperties()
{
 TileMapPropertyFinder finder = new TileMapPropertyFinder (currentLevel);
 foreach (var propertyLocation in finder.GetPropertyLocations())
 {
  var properties = propertyLocation.Properties;
  if (properties.ContainsKey ("EntityType"))
  {
   float worldX = propertyLocation.WorldX;
   float worldY = propertyLocation.WorldY;
   if (properties.ContainsKey ("YOffset"))
   {
    string yOffsetAsString = properties ["YOffset"];
    float yOffset = 0;
    float.TryParse (yOffsetAsString, out yOffset);
    worldY += yOffset;
   }
   bool created = TryCreateEntity (properties ["EntityType"], worldX, worldY);
   ...
  }
...
 }
}
```

Animated entities

Coin Time includes several animated entities. The `Player` and `Enemy` entities play walk animations and the `Door` entity plays an opening animation once all coins have been collected.

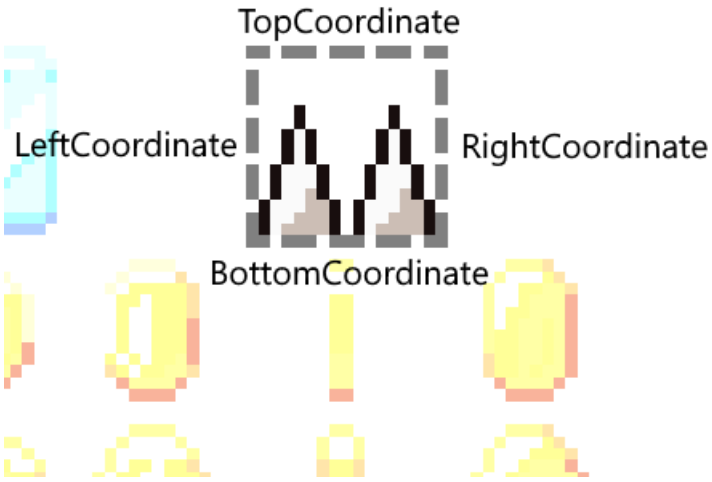
.achx files

Coin Time animations are defined in .achx files. Each animation is defined between `AnimationChain` tags, as shown in the following animation defined in **propanimations.achx**:

```
<AnimationChain>
  <Name>Spikes</Name>
  <ColorKey>0</ColorKey>
  <Frame>
    <FlipHorizontal>false</FlipHorizontal>
    <FlipVertical>false</FlipVertical>
    <TextureName>..\images\mastersheet.png</TextureName>
    <FrameLength>0.1</FrameLength>
    <LeftCoordinate>1152</LeftCoordinate>
    <RightCoordinate>1168</RightCoordinate>
    <TopCoordinate>128</TopCoordinate>
    <BottomCoordinate>144</BottomCoordinate>
    <RelativeX>0</RelativeX>
    <RelativeY>0</RelativeY>
  </Frame>
</AnimationChain>
```

This animation only contains a single frame, resulting in the Spike entity displaying a static image. Entities can use .achx files whether they display single or multi-frame animations. Additional frames can be added to .achx files without requiring any changes in code.

Frames define which image to display in the `TextureName` parameter, and the coordinates of the display in the `LeftCoordinate`, `RightCoordinate`, `TopCoordinate`, and `BottomCoordinate` tags. These represent the pixel coordinates of the frame of animation in the image which is being used – **mastersheet.png** in this case.



The `FrameLength` property defines the number of seconds that a frame in an animation should be displayed. Single-frame animations ignore this value.

All other AnimationChain properties in the .achx file are ignored by Coin Time.

AnimatedSpriteEntity

Animation logic is contained in the `AnimatedSpriteEntity` class, which serves as the base class for most entities used in the `GameScene`. It provides the following functionality:

- Loading of `.achx` files
- Animation cache of loaded animations
- CCSprite instance for displaying the animation
- Logic for changing the current frame

The Spikes constructor provides a simple example of how to load and use animations:

```
public Spikes ()
{
 LoadAnimations ("Content/animations/propanimations.achx");
 CurrentAnimation = animations [0];
}
```

The **propAnimations.achx** only contains one animation, so the constructor accesses this animation by index. If a .achx file contains multiple animations, then animations can be referenced by name, as shown in the `Enemy` constructor:

```
walkLeftAnimation = animations.Find (item => item.Name == "WalkLeft");
walkRightAnimation = animations.Find (item => item.Name == "WalkRight");
```

Summary

This guide covers the implementation details of coin time. Coin Time is created to be a complete game, but is also a project which can be easily modified and expanded. Readers are encouraged to spend time making modifications to levels, adding new levels, and creating new entities to further understand how Coin Time is implemented.

Related links

- Game Project (sample)

Drawing geometry with CCDrawNode

10/3/2018 • 5 minutes to read • Edit Online

`CCDrawNode` provides methods for drawing primitive objects such as lines, circles, and triangles.

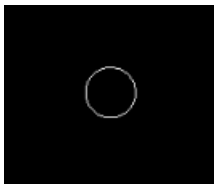
The `CCDrawNode` class in CocosSharp provides multiple methods for drawing common geometric shapes. It inherits from the `CCNode` class, and is usually added to `CCLayer` instances. This guide covers how to use `CCDrawNode` instances to perform custom rendering. It also provides a comprehensive list of available draw functions with screen shots and code examples.

Creating a CCDrawNode

The `CCDrawNode` class can be used to draw geometric objects such as circles, rectangles, and lines. For example, the following code sample shows how to create a `CCDrawNode` instance which draws a circle in a `CCLayer` implementing class:

```
public class GameLayer : CCLayer
{
    public GameLayer ()
    {
        var drawNode = new CCDrawNode ();
        this.AddChild (drawNode);
        // Origin is bottom-left of the screen. This moves
        // the drawNode 100 pixels to the right and 100 pixels up
        drawNode.PositionX = 100;
        drawNode.PositionY = 100;

        drawNode.DrawCircle (
            center: new CCPoint (0, 0),
            radius: 20,
            color: CCColor4B.White);

    }
}
```

This code produces the following circle at runtime:



Draw method details

Let's take a look at a few details related to drawing with a `CCDrawNode`:

Draw methods positions are relative to the CCDrawNode

All draw methods require at least one position value for drawing. This position value is relative to the `CCDrawNode` instance. This means that the `CCDrawNode` itself has a position, and all draw calls made on the `CCDrawNode` also take one or more position values. To help understand how these values combine, let's look at a few examples.

First we'll look at the `DrawCircle` example above:

```
...
drawNode.PositionX = 100;
drawNode.PositionY = 100;

drawNode.DrawCircle (center: new CCPoint (0, 0),
...
```

In this case, the `CCDrawNode` is positioned at (100,100), and the drawn circle is at (0,0) relative to the `CCDrawNode`, resulting in the circle being centered 100 pixels up and to the right of the bottom-left corner of the game screen.

The `CCDrawNode` can also be positioned at the origin (bottom left of the screen), relying on the circle for offsets:

```
...
drawNode.PositionX = 0;
drawNode.PositionY = 0;

drawNode.DrawCircle (center: new CCPoint (50, 60),
...
```

The code above results in the circle's center at 50 units (`drawNode.PositionX` + the `CCPoint.X`) to the right of the left side of the screen, and 60 (`drawNode.PositionY` + the `CCPoint.Y`) units above the bottom of the screen.

Once a draw method has been called, the drawn object cannot be modified unless the `CCDrawNode.Clear` method is called, so any repositioning needs to be done on the `CCDrawNode` itself.

Objects drawn by `CCNodes` are also impacted by the `CCNode` instance's `Rotation` and `Scale` properties.

Draw methods do not need to be called every frame

Draw methods need to be called only once to create a persistent visual. In the example above, the call to `DrawCircle` in the constructor of the `GameLayer` – `DrawCircle` does not need to be called every-frame to re-draw the circle when the screen refreshes.

This differs from draw methods in MonoGame, which typically will render something to the screen for only one frame, and which must be called every-frame.

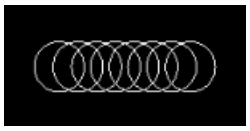
If draw methods are called every frame then objects will eventually accumulate inside the calling `CCDrawNode` instance, resulting in a drop in framerate as more objects are drawn.

Each CCDrawNode supports multiple draw calls

`CCDrawNode` instances can be used to draw multiple shapes. This allows complex visual objects to be encased in a single object. For example, the following code can be used to render multiple circles with one `CCDrawNode`:

```
for (int i = 0; i < 8; i++)
{
    drawNode.DrawCircle (
        center: new CCPoint (i*15, 0),
        radius: 20,
        color: CCColor4B.White);
}
```

This results in the following graphic:



Draw call examples

The following draw calls are available in `CCDrawNode`:

- `DrawCatmullRom`
- `DrawCircle`
- `DrawCubicBezier`
- `DrawEllipse`
- `DrawLineList`
- `DrawPolygon`
- `DrawQuadBezier`
- `DrawRect`
- `DrawSegment`
- `DrawSolidArc`
- `DrawSolidCircle`
- `DrawTriangleList`

DrawCardinalSpline

`DrawCardinalSpline` creates a curved line through a variable number of points.

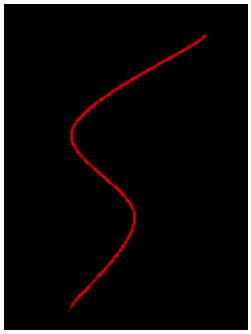
The `config` parameter defines which points the spline will pass through. The example below shows a spline which passes through four points.

The `tension` parameter controls how sharp or round the points on the spline appear. A `tension` value of 0 will result in a curved spline, and a `tension` value of 1 will result in a spline drawn by straight lines and hard edges.

Although splines are curved lines, CocosSharp draws splines with straight lines. The `segments` parameter controls how many segments to use to draw the spline. A larger number results in a smoothly curved spline at the cost of performance.

Code example:

```
var splinePoints = new List<CCPoint> ();
splinePoints.Add (new CCPoint (0, 0));
splinePoints.Add (new CCPoint (50, 70));
splinePoints.Add (new CCPoint (0, 140));
splinePoints.Add (new CCPoint (100, 210));

drawNode.DrawCardinalSpline (
    config: splinePoints,
    tension: 0,
    segments: 64,
    color:CCColor4B.Red);
```

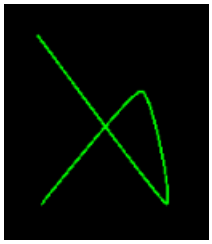


DrawCatmullRom

`DrawCatmullRom` creates a curved line through a variable number of points, similar to `DrawCardinalLine`. This method does not include a tension parameter.

Code example:

```
var splinePoints = new List<CCPoint> ();
splinePoints.Add (new CCPoint (0, 0));
splinePoints.Add (new CCPoint (80, 90));
splinePoints.Add (new CCPoint (100, 0));
splinePoints.Add (new CCPoint (0, 130));

drawNode.DrawCatmullRom (
    points: splinePoints,
    segments: 64);
```



DrawCircle

`DrawCircle` creates a perimeter of a circle of a given `radius`.

Code example:

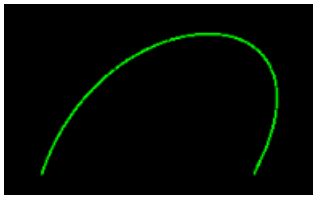
```
drawNode.DrawCircle (
    center:new CCPoint (0, 0),
    radius:20,
    color:CCColor4B.Yellow);
```

DrawCubicBezier

`DrawCubicBezier` draws a curved line between two points, using control points to set the path between the two points.

Code example:

```
drawNode.DrawCubicBezier (
    origin: new CCPoint (0, 0),
    control1: new CCPoint (50, 150),
    control2: new CCPoint (250, 150),
    destination: new CCPoint (170, 0),
    segments: 64,
    lineWidth: 1,
    color: CCColor4B.Green);
```

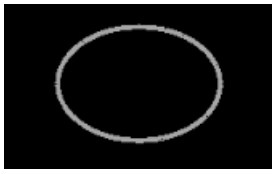


DrawEllipse

`DrawEllipse` creates the outline of an *ellipse*, which is often referred to as an oval (although the two are not geometrically identical). The shape of the ellipse can be defined by a `CCRect` instance.

Code example:

```
drawNode.DrawEllipse (
    rect: new CCRect (0, 0, 130, 90),
    lineWidth: 2,
    color: CCColor4B.Gray);
```



DrawLine

`DrawLine` connects to points with a line of a given width. This method is similar to `DrawSegment`, except it creates flat endpoints as opposed to round endpoints.

Code example:

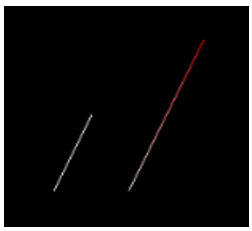
```
drawNode.DrawLine (
    from: new CCPoint (0, 0),
    to: new CCPoint (150, 30),
    lineWidth: 5,
    color:CCColor4B.Orange);
```

DrawLineList

`DrawLineList` creates multiple lines by connecting each pair of points specified by a `CCV3F_C4B` array. The `CCV3F_C4B` struct contains values for position and color. The `verts` parameter should always contain an even number of points, as each line is defined by two points.

Code example:

```
CCV3F_C4B[] verts = new CCV3F_C4B[] {
    // First line:
    new CCV3F_C4B( new CCPoint(0,0), CCColor4B.White),
    new CCV3F_C4B( new CCPoint(30,60), CCColor4B.White),
    // second line, will blend from white to red:
    new CCV3F_C4B( new CCPoint(60,0), CCColor4B.White),
    new CCV3F_C4B( new CCPoint(120,120), CCColor4B.Red)
};

drawNode.DrawLineList (verts);
```

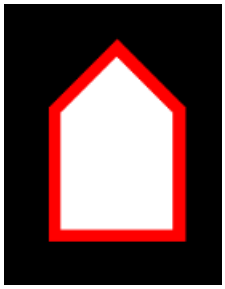


DrawPolygon

`DrawPolygon` creates a filled-in polygon with an outline of variable width and color.

Code example:

```
CCPoint[] verts = new CCPoint[] {
    new CCPoint(0,0),
    new CCPoint(0, 100),
    new CCPoint(50, 150),
    new CCPoint(100, 100),
    new CCPoint(100, 0)
};

drawNode.DrawPolygon (verts,
    count: verts.Length,
    fillColor: CCColor4B.White,
    borderWidth: 5,
    borderColor: CCColor4B.Red,
    closePolygon: true);
```

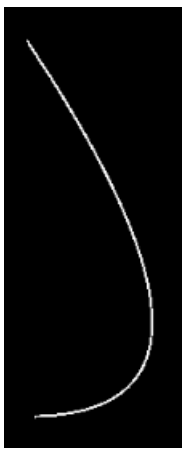


DrawQuadBezier

`DrawQuadBezier` connects two points with a line. It behaves similarly to `DrawCubicBezier` but only supports a single control point.

Code example:

```
drawNode.DrawQuadBezier (
    origin:new CCPoint (0, 0),
    control:new CCPoint (200, 0),
    destination:new CCPoint (0, 300),
    segments:64,
    lineWidth:1,
    color:CCColor4B.White);
```

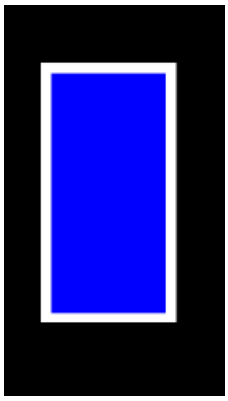


DrawRect

`DrawRect` creates a filled-in rectangle with an outline of variable width and color.

Code example:

```
var shape = new CCRect (
    0, 0, 100, 200);
drawNode.DrawRect(shape,
    fillColor:CCColor4B.Blue,
    borderWidth: 4,
    borderColor:CCColor4B.White);
```



DrawSegment

`DrawSegment` connects two points with a line of variable width and color. It is similar to `DrawLine`, except it creates round endpoints rather than flat endpoints.

Code example:

```
drawNode.DrawSegment (from: new CCPoint (0, 0),
    to: new CCPoint (100, 200),
    radius: 5,
    color:new CCColor4F(1,1,1,1));
```

DrawSolidArc

`DrawSolidArc` creates a filled-in wedge of a given color and radius.

Code example:

```
drawNode.DrawSolidArc(
    pos:new CCPoint(100, 100),
    radius:200,
    startAngle:0,
    sweepAngle:CCMathHelper.Pi/2, // this is in radians, clockwise
    color:CCColor4B.White);
```

DrawSolidCircle

`DrawCircle` creates a filled-in circle of a given radius.

Code example:

```
drawNode.DrawSolidCircle(
    pos: new CCPoint (100, 100),
    radius: 50,
    color: CCColor4B.Yellow);
```

DrawTriangleList

`DrawTriangleList` creates a list of triangles. Each triangle is defined by three `CCV3F_C4B` instances in an array. The number of vertices in the array passed to the `verts` parameter must be a multiple of three. Note that the only information contained in `CCV3F_C4B` is the position of the verts and their color – the `DrawTriangleList` method does not support drawing triangles with textures.

Code example:

```
CCV3F_C4B[] verts = new CCV3F_C4B[] {
    // First triangle:
    new CCV3F_C4B( new CCPoint(0,0), CCColor4B.White),
    new CCV3F_C4B( new CCPoint(30,60), CCColor4B.White),
    new CCV3F_C4B( new CCPoint(60,0), CCColor4B.White),
    // second triangle, each point has different colors:
    new CCV3F_C4B( new CCPoint(90,0), CCColor4B.Yellow),
    new CCV3F_C4B( new CCPoint(120,60), CCColor4B.Red),
    new CCV3F_C4B( new CCPoint(150,0), CCColor4B.Blue)
};

drawNode.DrawTriangleList (verts);
```

Summary

This guide explains how to create a `CCDrawNode` and perform primitive-based rendering. It provides an example of each of the draw calls.

Related links

- CCDrawNode API
- Full Sample

Animating with CCAction

10/3/2018 • 7 minutes to read • Edit Online

The CCAction class simplifies adding animations to CocosSharp games. These animations can be used to implement functionality or to add polish.

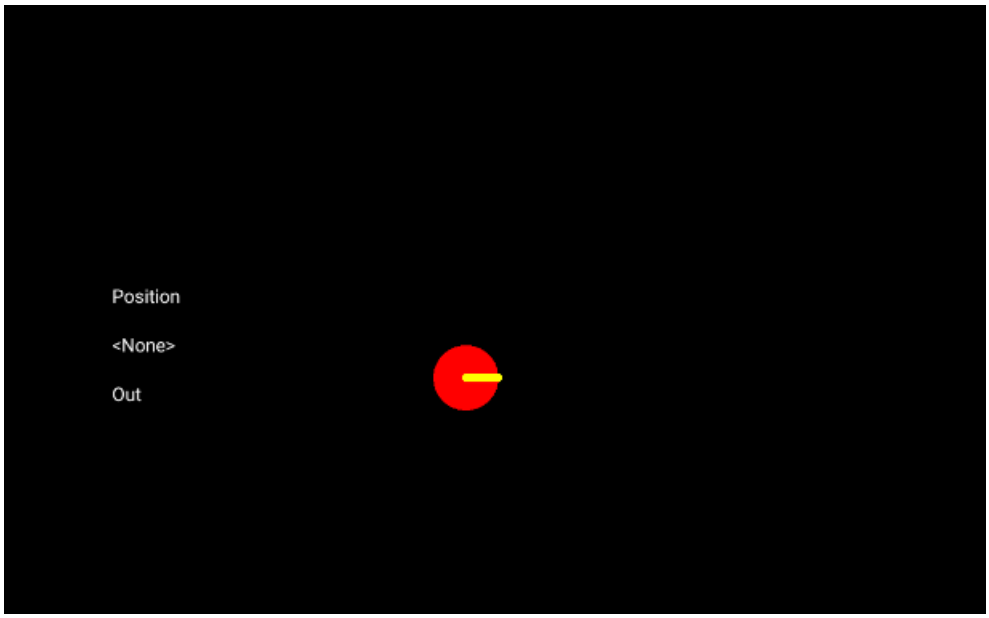
`CCAction` is a base class which can be used to animate CocosSharp objects. This guide covers built-in `CCAction` implementations for common tasks such as positioning, scaling, and rotating. It also looks at how to create custom implementations by inheriting from `CCAction`.

This guide uses a project called **ActionProject** which can be downloaded here. This guide uses the `CCDrawNode` class, which is covered in the Drawing Geometry with CCDrawNode guide.

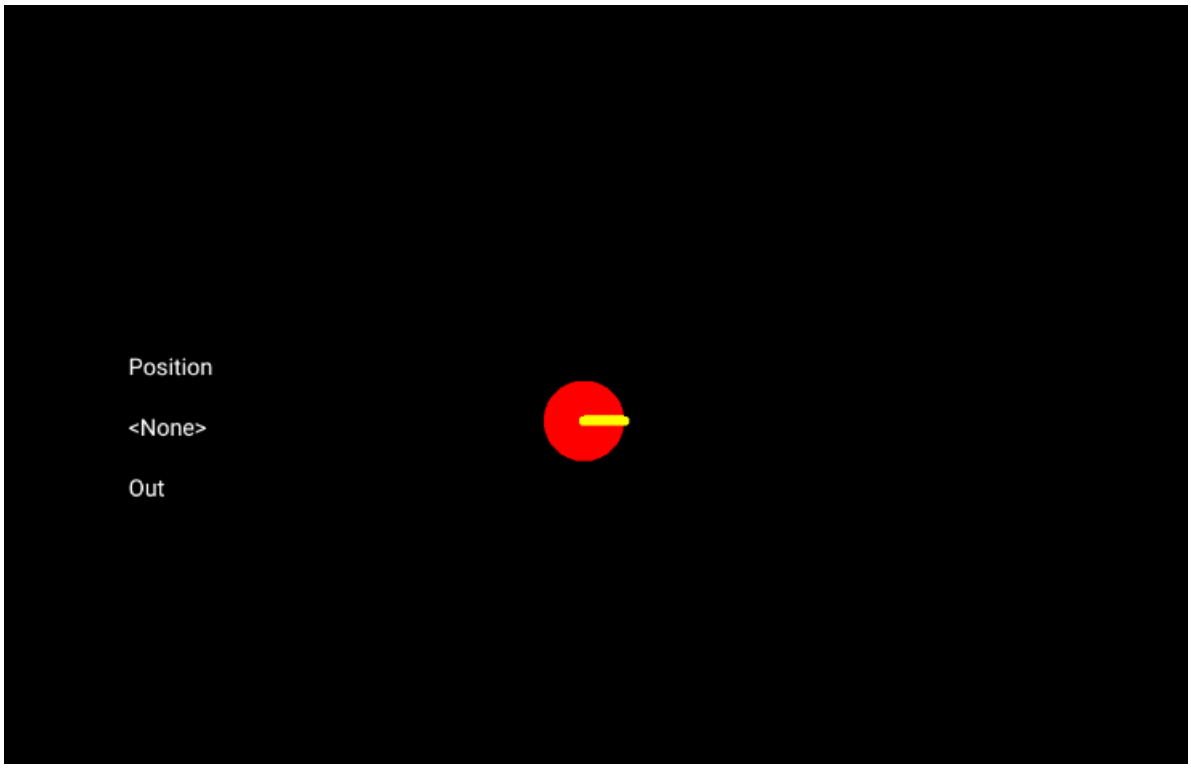
Running the ActionProject

ActionProject is a CocosSharp solution which can be built for iOS and Android. It serves both as a code sample for how to use the `CCAction` class and as a real-time demo of common `CCAction` implementations.

When running, ActionProject displays three `CCLabel` instances on the left of the screen and a visual object drawn by two `CCDrawNode` instances for viewing the various actions:



The labels on the left indicate which type of `CCAction` will be created when tapping on the screen. By default, the **Position** value is selected, resulting in a `CCMove` action being created and applied to the red circle:



Clicking the labels on the left changes which type of `CCAction` is performed on the circle. For example, clicking the **Position** label will cycle through the different values that can be changed:

Common variable-changing CCAction classes

The **ActionProject** uses the following `CCAction`-inheriting classes, which are a part of CocosSharp:

- `CCMoveTo` – Changes a `CCNode` instance's `Position` property
- `CCScaleTo` – Changes a `CCNode` instance's `Scale` property
- `CCRotateTo` – Changes a `CCNode` instance's `Rotation` property

This guide refers to these actions as *variable-changing*, which means that they directly impact the variable of the `CCNode` that they are added to. Other types of actions are referred to as *easing* actions, which are covered later in

this guide.

The **ActionProject** demonstrates that the purpose of these actions is to modify a variable over time. Specifically, these `CCActions` constructors take two arguments: length of time to take and the value to assign. The following piece of code shows how the three types of actions are created:

```
switch (VariableOptions [currentVariableIndex])
{
    case "Position":
        coreAction = new CCMoveTo(timeToTake, touch.Location);

        break;
    case "Scale":
        var distance = CCPoint.Distance (touch.Location, drawNodeRoot.Position);
        var desiredScale = distance / DefaultCircleRadius;
        coreAction = new CCScaleTo(timeToTake, desiredScale);

        break;
    case "Rotation":
        float differenceY = touch.Location.Y - drawNodeRoot.PositionY;
        float differenceX = touch.Location.X - drawNodeRoot.PositionX;

        float angleInDegrees = -1 * CCMathHelper.ToDegrees(
            (float)System.Math.Atan2(differenceY, differenceX));

        coreAction = new CCRotateTo (timeToTake, angleInDegrees);

        break;
...
}
```

Once the action is created, it is added to a CCNode as follows:

```
nodeToAddTo.AddAction (coreAction);
```

`AddAction` starts the `CCAction` instance's behavior, and it will automatically perform its logic frame-after-frame until completion.

Each of the types listed above ends with the word *To* which means the `CCAction` will modify the `CCNode` so that the argument value represents the final state when the action has finished. For example, creating a `CCMoveTo` with a position of X = 100 and Y = 200 results in the `CCNode` instance's `Position` being set to X = 100, Y = 200 at the end of the time specified, regardless of the `CCNode` instance's starting location.

Each "To" class also has a "By" version, which will add the argument value to the current value on the `CCNode`. For example, creating a `CCMoveBy` with a position of X = 100 and Y = 200 will result in the `CCNode` instance being moved to the right 100 units and up 200 units from the position it was at when the action was started.

Easing actions

By default, variable-changing actions will perform *linear interpolation* – the action will move towards the desired value at a constant rate. If interpolating *position* linearly, the moving object will immediately start and stop moving at the beginning and end of the action, and its speed will stay constant as the action executes.

Non-linear interpolation is less jarring and adds an element of polish, so CocosSharp offers a variety of easing actions which can be used to modify variable-changing actions.

In the **ActionProject** sample, we can switch between these types of easing actions by clicking on the second label (which defaults to):

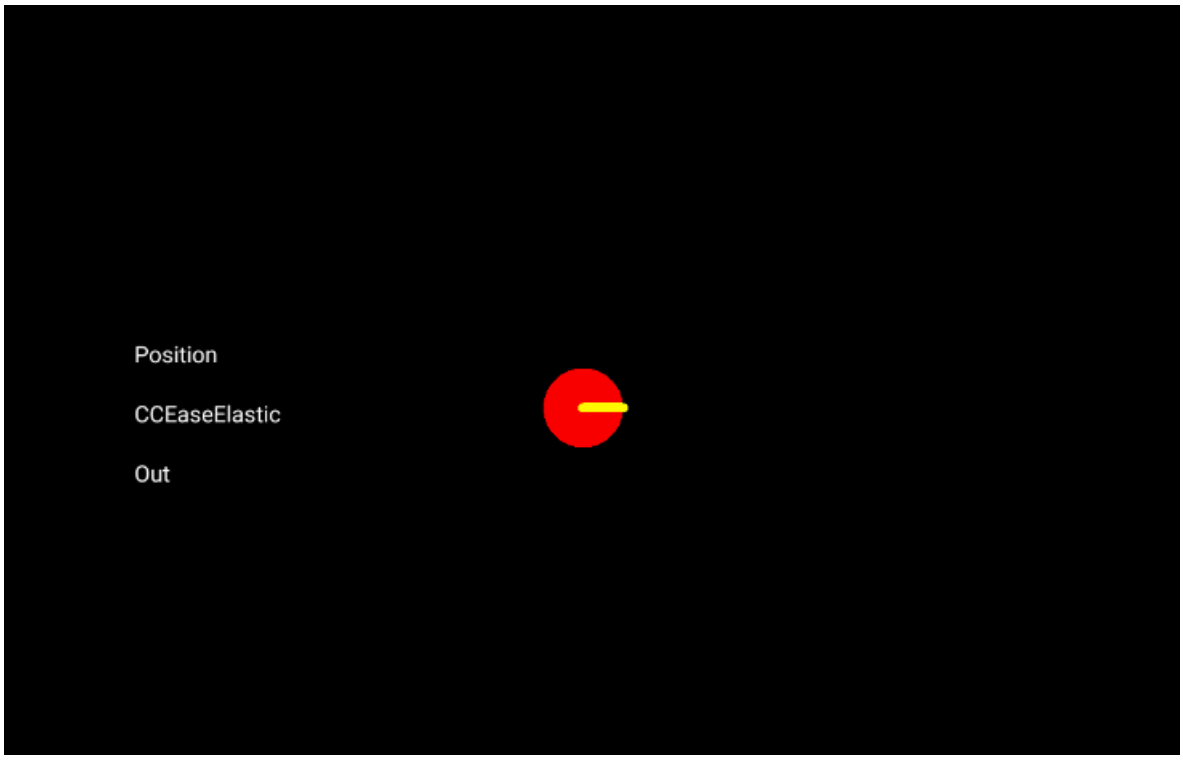
Easing actions are especially powerful because they are not tied to any particular variable-setting action. This means that the same easing action can be used to assign position, rotation, scale, or custom actions (as will be shown later in this guide).

Easing actions wrap variable-setting actions (so long as the variable-setting action inherits from `CCFiniteTimeAction`) by accepting a variable-setting action as an argument in their constructors.

For example, if the labels are set to **Position**, **CCEaseElastic**, then the following code will execute when a touch is detected (note that code has been omitted to highlight the relevant lines):

```
CCFiniteTimeAction coreAction = null;
...
coreAction = new CCMoveTo(timeToTake, touch.Location);
...
CCAction easing = null;
...
easing = new CCEaseSineOut (coreAction);
...
nodeToAddTo.AddAction (easing);
```

As shown by the application, the exact same easing can be applied to other variable-setting actions such as `CCRotateTo`:



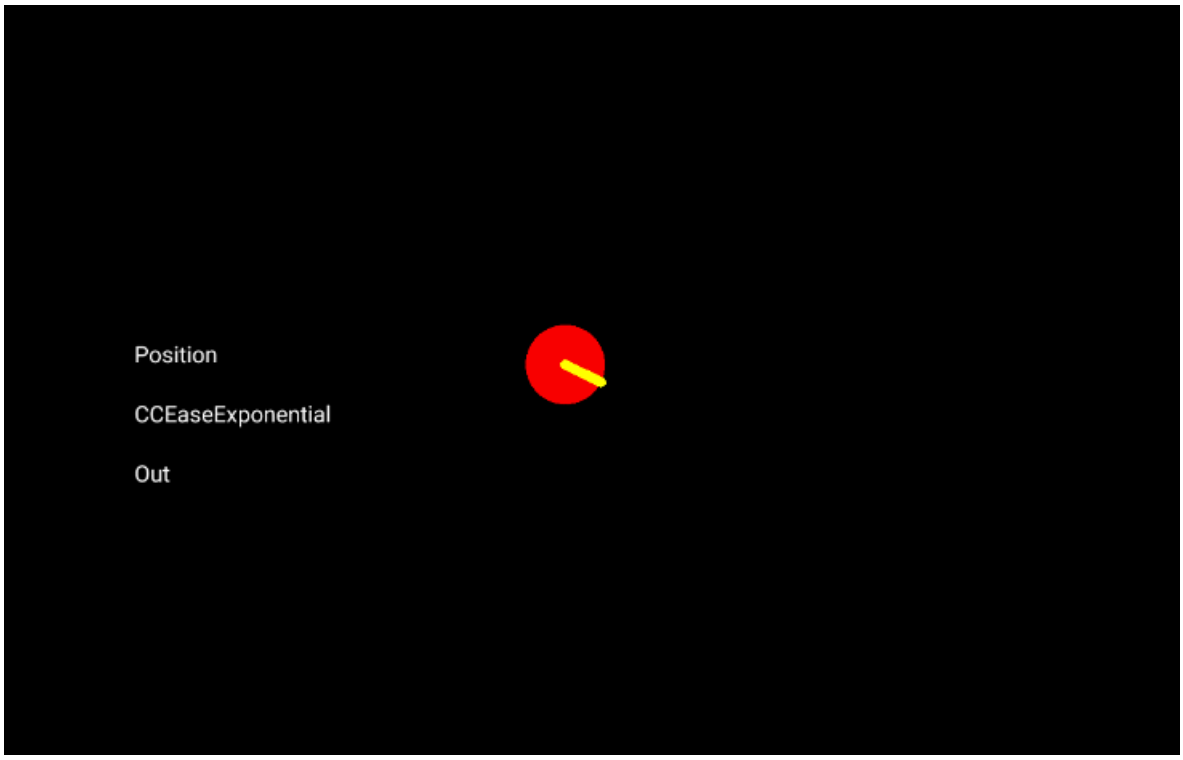
Easing In, Out, and InOut

All easing actions have `In`, `Out`, or `InOut` appended to the easing type. These terms refer to when the easing is applied: `In` means easing will be applied at the beginning, `Out` means at the end, and `InOut` means both at the beginning and end.

An `In` easing action will impact the way a variable is applied throughout the entire interpolation (both at the beginning and at the end), but typically the most recognizable characteristics of the easing action will take place at the beginning. Similarly, `Out` easing actions are characterized by their behavior at the end of the interpolation. For example, `CCEaseBounceOut` will result in an object bouncing at the end of the action.

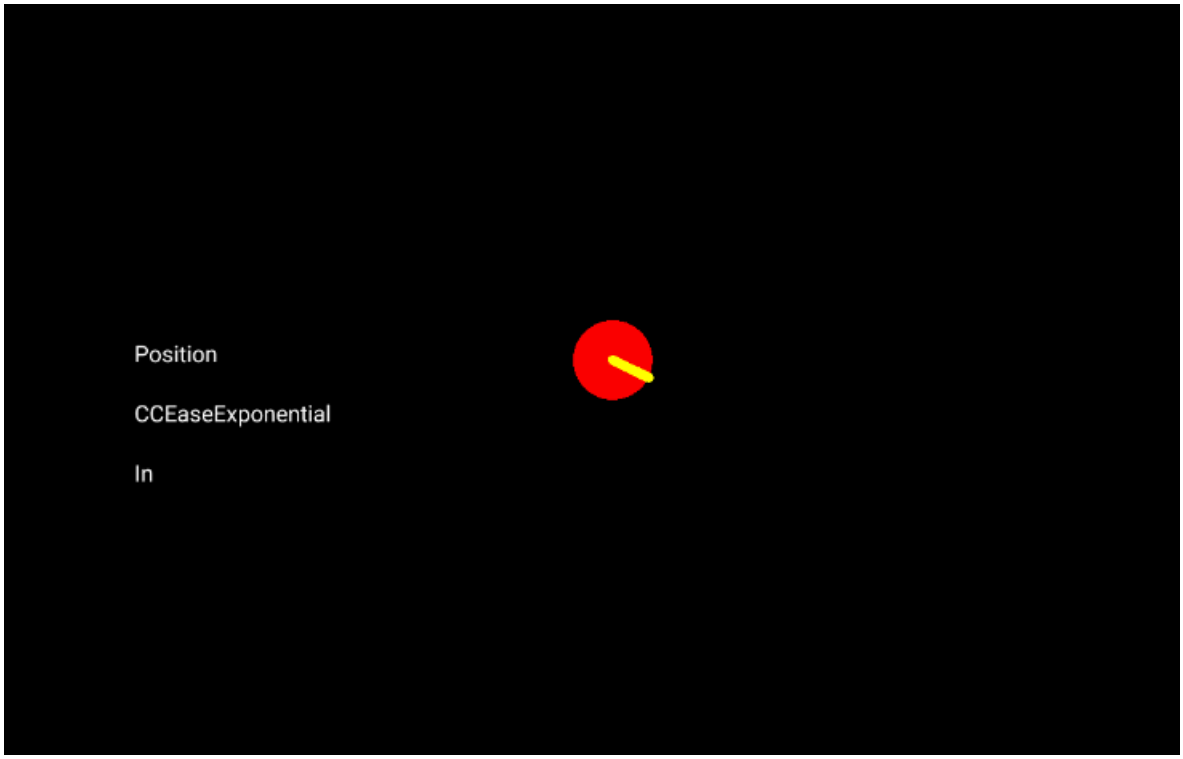
Out

`Out` easing generally applies the most noticeable changes at the end of the interpolation. For example, `CCEaseExponentialOut` will slow the rate of change of the changing variable as it approaches the target value:



In

`In` easing generally applies the most noticeable change at the beginning of the interpolation. For example, `CCEaseExponentialIn` will move more slowly at the beginning of the action:



InOut

`InOut` generally applies the most noticeable changes both at the beginning and end. `InOut` easing is usually symmetric. For example, `CCEaseExponentialInOut` will move slowly at the beginning and end of the action:

Implementing a custom CCAction

All of the classes we've discussed so far are included in CocosSharp to provide common functionality. Custom `CCAction` implementations can provide additional flexibility. For example, a `CCAction` which controls the filled ratio of an experience bar can be used so that the experience bar grows smoothly whenever the user earns experience.

`CCAction` implementations typically require two classes:

- `CCFiniteTimeAction` implementation – The finite time action class is responsible for starting the action. It is the class which is instantiated and either added directly to a `CCNode` or to an easing action. It must instantiate and return a `CCFiniteTimeActionState`, which will perform updates.
- `CCFiniteTimeActionState` implementation – The finite time action state class is responsible for updating the variables involved in the action. It must implement an Update function, which assigns the value on the target according to a time value. This class is not explicitly referenced outside of the `CCFiniteTimeAction` which creates it. It simply works "behind the scenes".

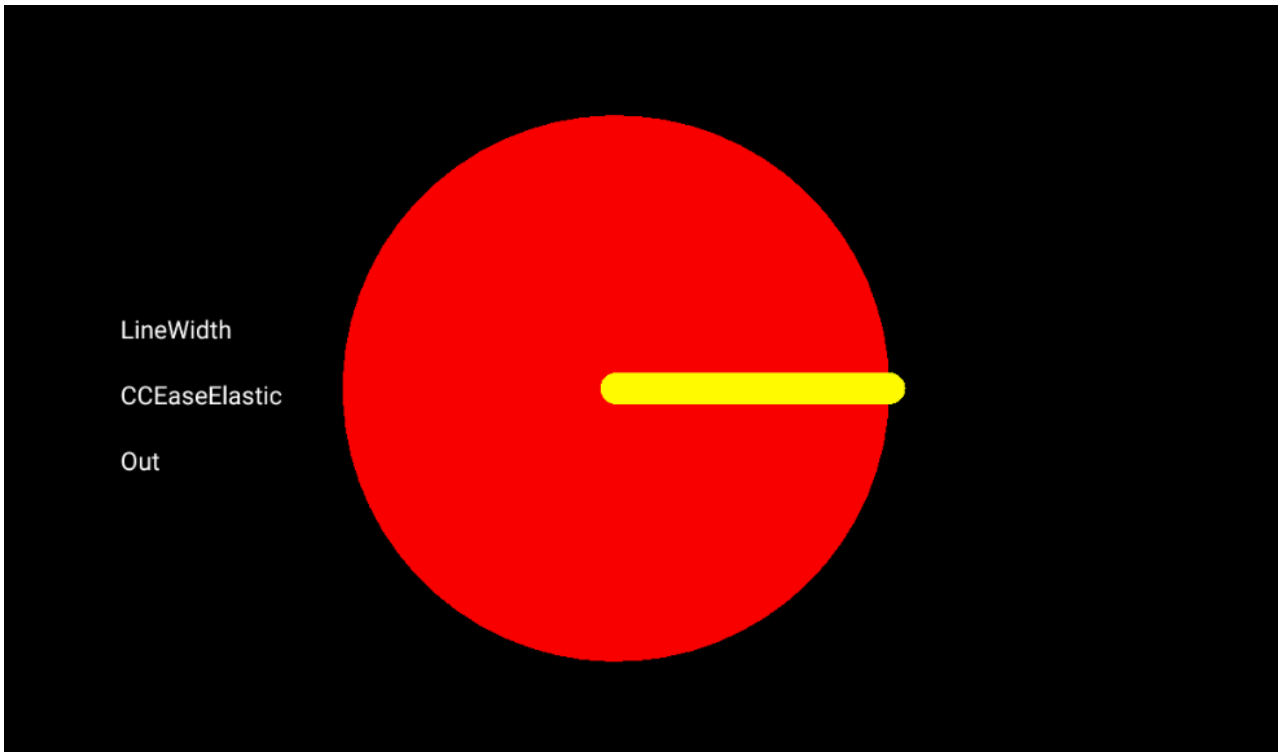
ActionProject provides a custom `CCFiniteTimeAction` implementation called `LineWidthAction,` which is used to adjust the width of the yellow line drawn on top of the red circle. Note that its only job is to instantiate and return a `LineWidthState` instance:

```
public class LineWidthAction : CCFiniteTimeAction
{
    float endWidth;

    public LineWidthAction (float duration, float width) : base(duration)
    {
        endWidth = width;
    }

    public override CCFiniteTimeAction Reverse ()
    {
        throw new NotImplementedException ();
    }

    protected override CCActionState StartAction (CCNode target)
    {
        return new LineWidthState (this, target, endWidth);
    }
}
```

As mentioned above, the `LineWidthState` does the work of assigning the line's `Width` property according to how much `time` has passed:

```
public class LineWidthState : CCFiniteTimeActionState
{
    float deltaWidth;
    float startWidth;

    LineNode castedTarget;

    public LineWidthState(LineWidthAction action, CCNode target, float endWidth) : base(action, target)
    {
        castedTarget = target as LineNode;

        if (castedTarget == null)
        {
            throw new InvalidOperationException ("The argument target must be a LineNode");
        }

        startWidth = castedTarget.Width;
        deltaWidth = endWidth - startWidth;
    }

    public override void Update (float time)
    {
        castedTarget.Width = startWidth + deltaWidth * time;
    }
}
```

The LineWidthAction can be combined with any easing action to change the line width in various ways, as shown in the following animation:



Interpolation and the Update method

The only logic, aside from storing values in the classes above, lives in the `LineWidthState.Update` method. The `startWidth` variable stores the width of the target `LineNode` at the start of the action, and the `deltaWidth` variable stores how much the value will change over the course of the action.

By substituting the `time` variable with a value of 0, we can see that the target `LineNode` will be at its starting position:

```
castedTarget.Width = startWidth + deltaWidth * 0;
```

Similarly, we can see that the target `LineNode` will be at its destination by substituting the time variable with a value of 1:

```
castedTarget.Width = startWidth + deltaWidth * 1;
```

The `time` value will usually be between 0 and 1 - but not always - and `Update` implementations should not assume these bounds. Some easing methods (such as `CCEaseBackIn` and `CCEaseBackOut`) will provide a time value outside of the 0 to 1 range.

Conclusion

Interpolation and easing are a critical part of creating a polished game, especially when creating user interfaces. This guide covers how to use `CCActions` to interpolate standard values such as position and rotation as well as custom values. The `LineWidthState` and `LineWidthAction` classes show how to implement a custom action.

Related links

- CCAction
- CCMoveTo
- CCScaleTo
- CCRotateTo

- CCDrawNode
- Full Sample

Using Tiled with CocosSharp

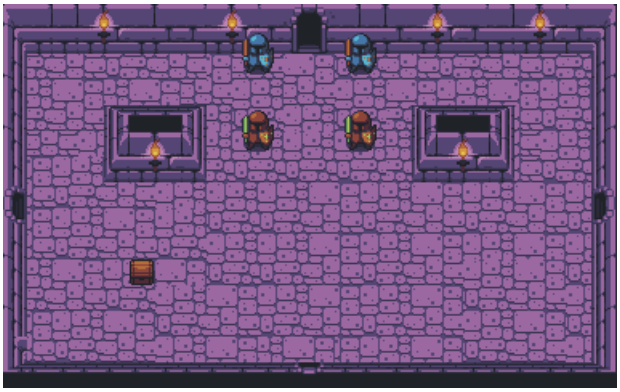
10/3/2018 • 6 minutes to read • Edit Online

Tiled is a powerful, flexible, and mature application for creating orthogonal and isometric tile maps for games. CocosSharp provides built-in integration for Tiled's native file format.

The *Tiled* application is a standard for creating *tile maps* for use in game development. This guide will walk through how to take an existing .tmx file (file created by Tiled) and use it in a CocosSharp game. Specifically this guide will cover:

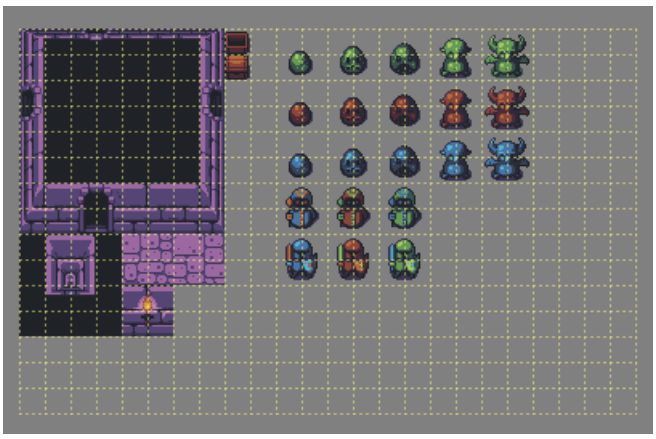
- The purpose of tile maps
- Working with .tmx files
- Considerations for rendering pixel art
- Using Tile properties at runtime

When finished we will have the following demo:



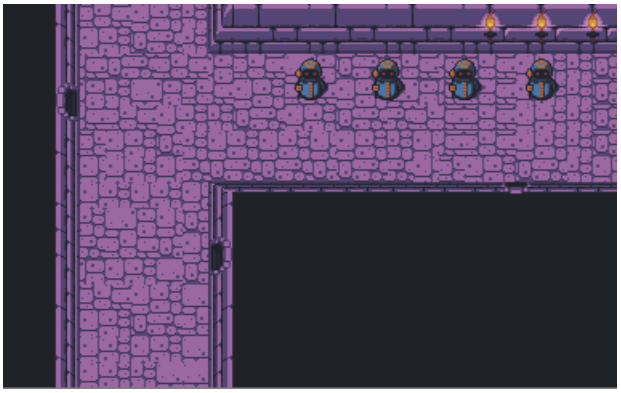
The purpose of tile maps

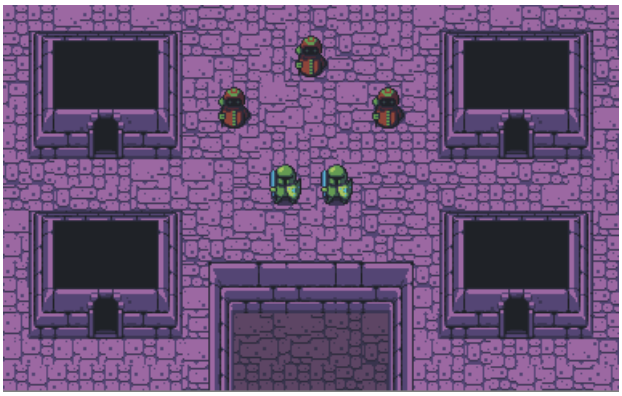
Tile maps have existed in game development for decades, but are still commonly used in 2D games for their efficiency and esthetics. Tile maps are able to achieve a very high level of efficiency through their use of tile sets – the source image used by tile maps. A tile set is a collection of images combined into one file. Although tile sets refer to images used in tile maps, files that contain multiple smaller images are also called sprite sheets or sprite maps in game development. We can visualize how tile sets are used by adding a grid to the tile set that we'll be using in our demo:



Tile maps arrange the individual tiles from tile sets. We should note that each tile map does not need to store its own copy of the tile set – rather, multiple tile maps can reference the same tile set. This means that aside from the

tile set, tile maps require very little memory. This enables the creation of a large number of tile maps, even when they are used to create a large game play area, such as a scrolling platformer environment. The following shows possible arrangements using the same tile set:



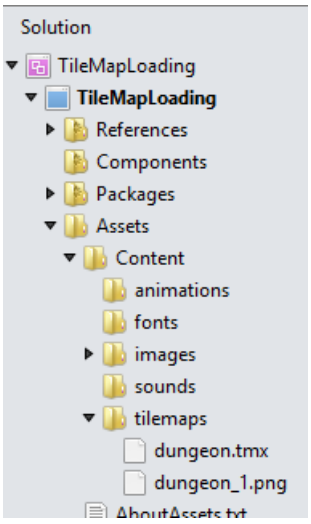


Working with .tmx files

The .tmx file format is an XML file created by the Tiled application, which can be downloaded for free on the Tiled website. The .tmx file format stores the information for tile maps. Typically a game will have one .tmx file for each level or separate area.

This guide focuses on how to use existing .tmx files in CocosSharp; however, additional tutorials can be found online, including this introduction to the Tiled map editor.

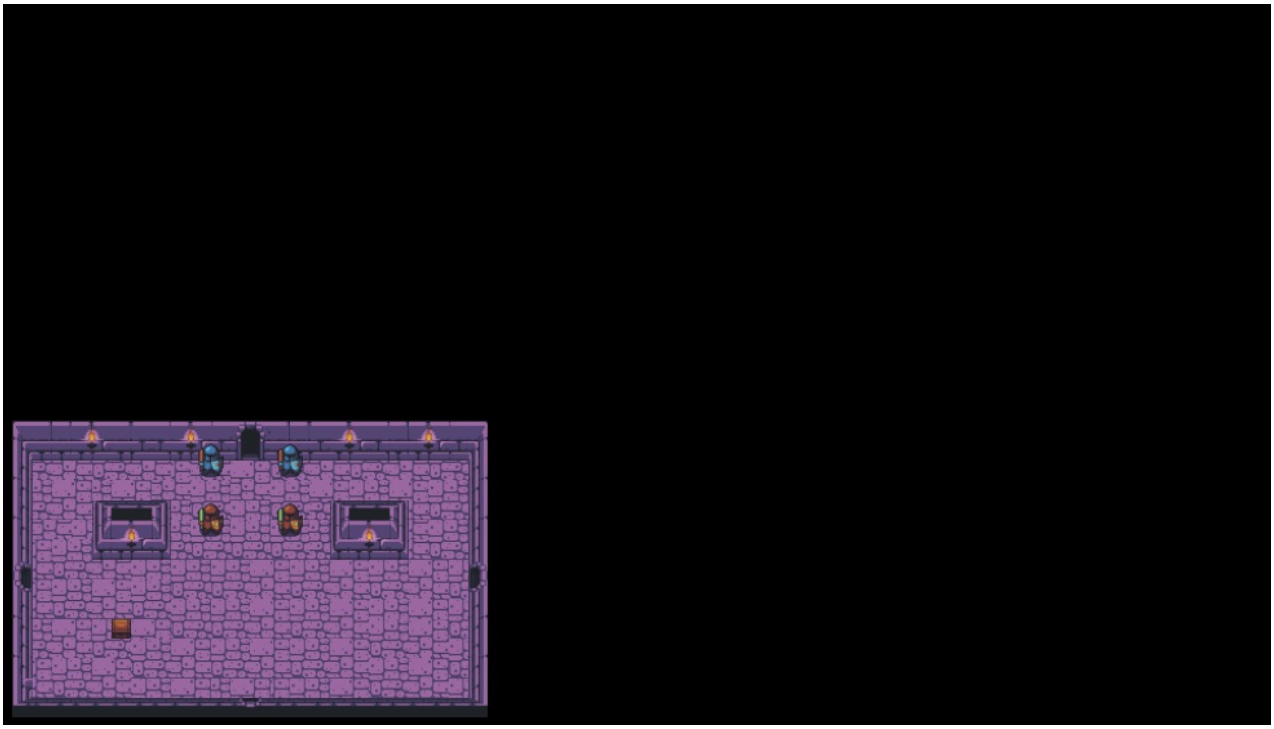
You'll need to unzip the content zip file to use it in our game. The first thing to note is that tile maps use both the .tmx file (dungeon.tmx) as well as one or more image files which define the tile set data (dungeon_1.png). The game needs to include both of these files to load the .tmx at runtime, so add both to the project's **Content** folder (which is contained in the **Assets** folder in Android projects). Specifically, add the files to a folder called **tilemaps** inside the **Content** folder:



The **dungeon.tmx** file can now be loaded at runtime into a `CCTileMap` object. Next, modify the `GameLayer` (or equivalent root container object) to contain a `CCTileMap` instance and to add it as a child:

```
public class GameLayer : CCLayer
{
    CCTileMap tileMap;

    public GameLayer ()
    {
    }

    protected override void AddedToScene ()
    {
        base.AddedToScene ();

        tileMap = new CCTileMap ("tilemaps/dungeon.tmx");
        this.AddChild (tileMap);
    }
}
```

If we run the game we will see the tile map appear in the bottom-left corner of the screen:



Considerations for rendering pixel art

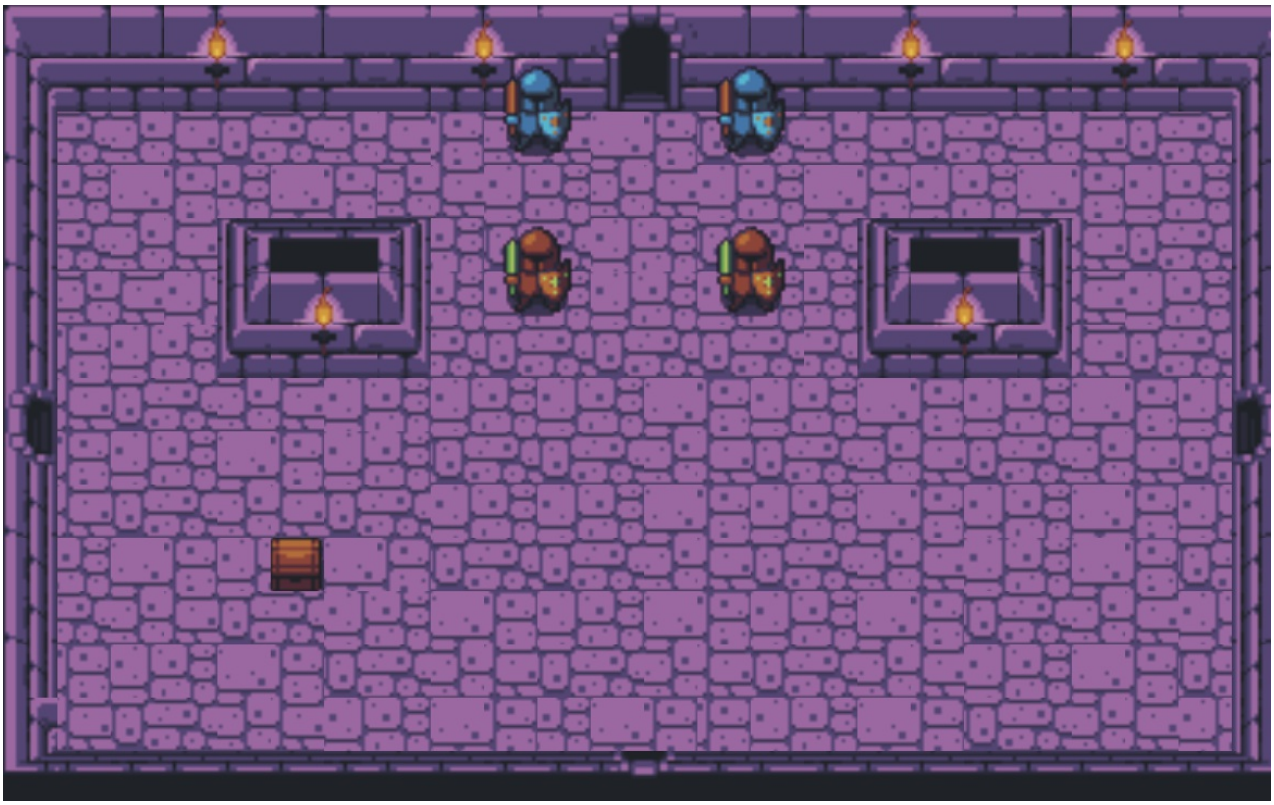
Pixel art, in the context of video game development, refers to 2D visual art which is typically created by-hand, and is often low resolution. Pixel art can be restrictively time intensive to create, so pixel art tile sets often include low-resolution tiles, such as 16 or 32 pixel width and height. If not scaled at runtime, pixel art is often too small for most modern phones and tablets.

We can adjust displayed dimensions in our game's GameAppDelegate.cs file, where we'll add a call to `CCScene.SetDefaultDesignResolution`:

```
 public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
    application.PreferMultiSampling = false;
    application.ContentRootDirectory = "Content";
    application.ContentSearchPaths.Add ("animations");
    application.ContentSearchPaths.Add ("fonts");
    application.ContentSearchPaths.Add ("sounds");

    CCSize windowSize = mainWindow.WindowSizeInPixels;

    float desiredWidth = 1024.0f;
    float desiredHeight = 768.0f;

    // This will set the world bounds to be (0,0, w, h)
    // CCSceneResolutionPolicy.ShowAll will ensure that the aspect ratio is preserved
    CCScene.SetDefaultDesignResolution (desiredWidth, desiredHeight, CCSceneResolutionPolicy.ShowAll);

    // Determine whether to use the high or low def versions of our images
    // Make sure the default texel to content size ratio is set correctly
    // Of course you're free to have a finer set of image resolutions e.g (ld, hd, super-hd)
    if (desiredWidth < windowSize.Width)
    {
        application.ContentSearchPaths.Add ("images/hd");
        CCSprite.DefaultTexelToContentSizeRatio = 2.0f;
    }
    else
    {
        application.ContentSearchPaths.Add ("images/ld");
        CCSprite.DefaultTexelToContentSizeRatio = 1.0f;
    }

    // New code:
    CCScene.SetDefaultDesignResolution (380, 240, CCSceneResolutionPolicy.ShowAll);

    CCScene scene = new CCScene (mainWindow);
    GameLayer gameLayer = new GameLayer ();

    scene.AddChild (gameLayer);

    mainWindow.RunWithScene (scene);
}
```

For more information on `CCSceneResolutionPolicy`, see our guide on handling resolutions in CocosSharp.

If we run the game now, we'll see the game take up the full screen of our device:

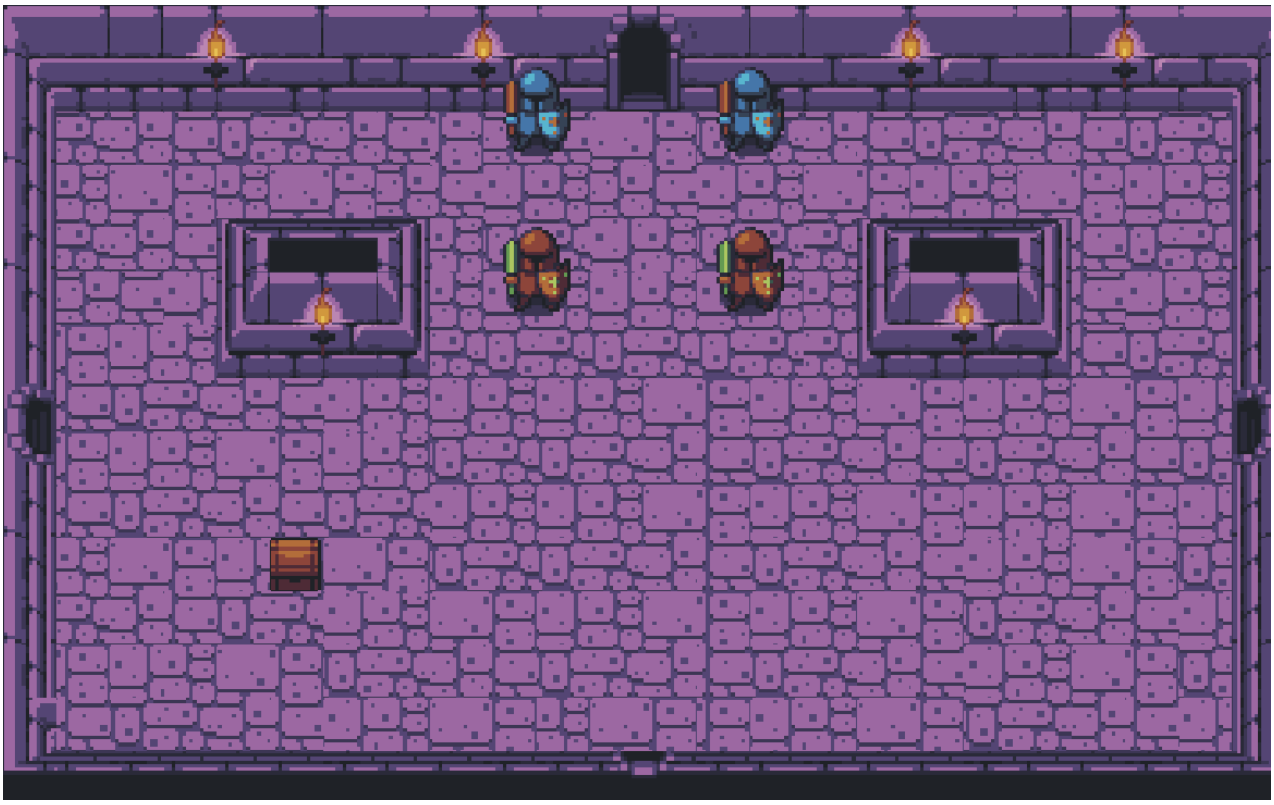


Finally we'll want to disable antialiasing on our tile map. The `Antialiased` property applies a blurring effect when rendering objects which are zoomed in. Antialiasing can be useful for reducing the pixelated look of graphical objects, but may also introduce its own rendering artifacts. Specifically, antialiasing blurs the contents of each tile. However, the edges of each tile are not blurred, which makes the individual tiles stand out rather than blending in with adjacent tiles. We should also note that pixel art games often preserve their pixelated look to maintain a *retro* feel.

Set `Antialiased` to `false` after constructing the `tileMap`:

```
protected override void AddedToScene ()
{
    base.AddedToScene ();

    tileMap = new CCTileMap ("tilemaps/dungeon.tmx");

    // new code:
    tileMap.Antialiased = false;

    this.AddChild (tileMap);
}
```

Now our tile map will not appear blurry:

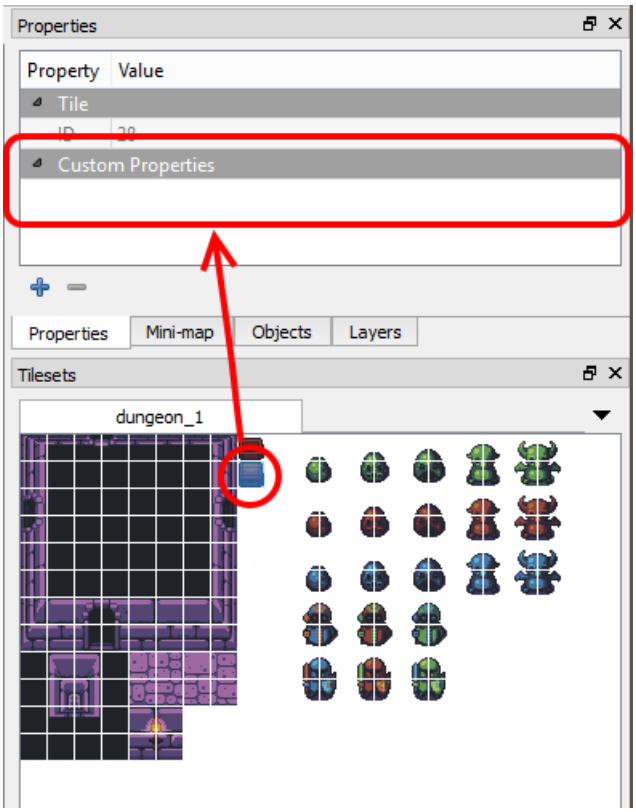


Using tile properties at runtime

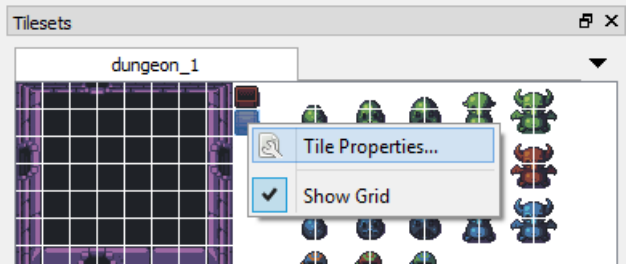
So far we have a `CCTileMap` loading a .tmx file and displaying it, but we have no way to interact with it. Specifically, certain tiles (such as our treasure chest) need to have custom logic. We'll step through how to detect custom tile properties, and various ways to react to these properties once identified at runtime.

Before we write any code, we'll need to add properties to our tile map through Tiled. To do this, open the dungeon.tmx file in the Tiled program. Be sure to open the file which is being used in the game project.

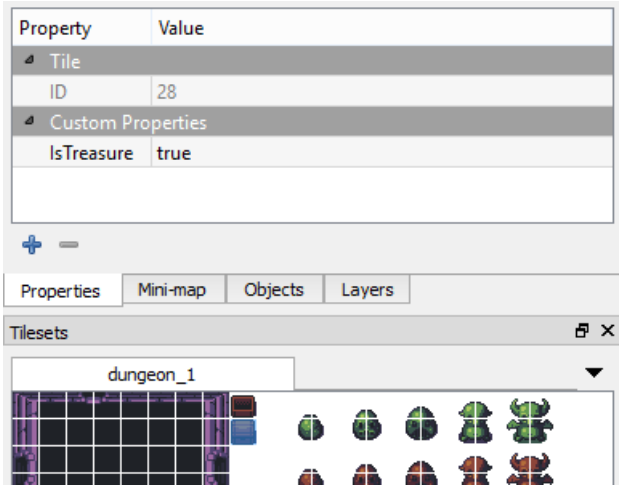
Once open, select the treasure chest in the tile set to view its properties:



If the treasure chest properties do not appear, right-click on the treasure chest and select **Tile Properties**:



Tiled properties are implemented with a name and a value. To add a property, click the **+** button, enter the name **IsTreasure**, click **OK**, then enter the value **true**:



Don't forget to save the .tmx file after modifying properties.

Finally, we'll add code to look for our newly-added property. We will loop through each `CCTileMapLayer` (our map has 2 layers), then through each row and column to look for any tiles which have the `IsTreasure` property:

```
public class GameLayer : CCLayer
{
    CCTileMap tileMap;

    public GameLayer ()
    {
    }

    protected override void AddedToScene ()
    {
        base.AddedToScene ();

        tileMap = new CCTileMap ("tilemaps/dungeon.tmx");

        // new code:
        tileMap.Antialiased = false;

        this.AddChild (tileMap);

        HandleCustomTileProperties (tileMap);
    }

    void HandleCustomTileProperties(CCTileMap tileMap)
    {
        // Width and Height are equal so we can use either
        int tileDimension = (int)tileMap.TileTexelSize.Width;

        // Find out how many rows and columns are in our tile map
        int numberOfColumns = (int)tileMap.MapDimensions.Size.Width;
        int numberOfRows = (int)tileMap.MapDimensions.Size.Height;
```

```
        // Tile maps can have multiple layers, so let's loop through all of them:
        foreach (CCTileMapLayer layer in tileMap.TileLayersContainer.Children)
        {
            // Loop through the columns and rows to find all tiles
            for (int column = 0; column < numberOfColumns; column++)
            {
                // We're going to add tileDimension / 2 to get the position
                // of the center of the tile - this will help us in
                // positioning entities, and will eliminate the possibility
                // of floating point error when calculating the nearest tile:
                int worldX = tileDimension * column + tileDimension / 2;
                for (int row = 0; row < numberOfRows; row++)
                {
                    // See above on why we add tileDimension / 2
                    int worldY = tileDimension * row + tileDimension / 2;

                    HandleCustomTilePropertyAt (worldX, worldY, layer);
                }
            }
        }
    }

    void HandleCustomTilePropertyAt(int worldX, int worldY, CCTileMapLayer layer)
    {
        CCTileMapCoordinates tileAtXy = layer.ClosestTileCoordAtNodePosition (new CCPoint (worldX, worldY));

        CCTileGidAndFlags info = layer.TileGIDAndFlags (tileAtXy.Column, tileAtXy.Row);

        if (info != null)
        {
            Dictionary<string, string> properties = null;

            try
            {
                properties = tileMap.TilePropertiesForGID (info.Gid);
            }
            catch
            {
                // CocosSharp
            }

            if (properties != null && properties.ContainsKey ("IsTreasure") && properties["IsTreasure"]
== "true" )
            {
                layer.RemoveTile (tileAtXy);

                // todo: Create a treasure chest entity
            }
        }
    }
}
```

Most of the code is self-explanatory, but we should discuss the handling of treasure tiles. In this case we are removing any tiles which are identified as treasure chests. This is because treasure chests will likely need custom code at runtime to effect collision, and to award the player the contents of the treasure when opened. Furthermore, the treasure may need to react to being opened (changing its visual appearance) and may have logic for only appearing when all on-screen enemies have been defeated.

In other words, the treasure chest will benefit from being an entity rather than being a simple tile in the `CCTileMap`. For more information on game entities, see the Entities in CocosSharp guide.

Summary

This walkthrough covers how to load .tmx files created by Tiled into a CocosSharp application. It shows how to

modify the app resolution to account for lower-resolution pixel art, and how to find tiles by their properties to perform custom logic, like creating entity instances.

Related links

- Tiled Website
- Content zip

Entities in CocosSharp

10/3/2018 • 11 minutes to read • Edit Online

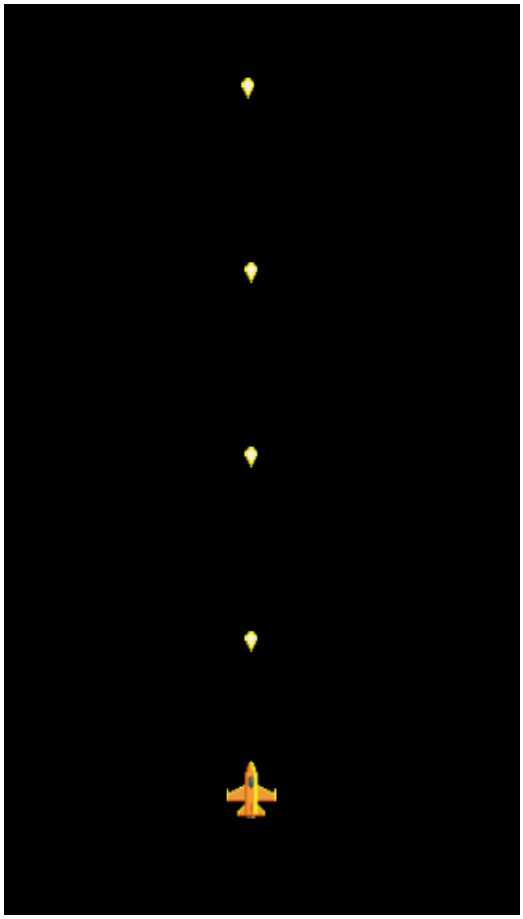
The entity pattern is a powerful way to organize game code. It improves readability, makes code easier to maintain, and leverages built-in parent/child functionality.

The entity pattern can improve a developer's efforts with CocosSharp through improved code organization. This walkthrough will be a hands-on example showing how to create two entities – a Ship entity and a Bullet entity. These entities will be self-contained objects, which means that once instantiated they will automatically be rendered and will perform movement logic as appropriate for their type.

This guide covers the following topics:

- Introduction to game entities
- General vs. specific entity types
- Project setup
- Creating entity classes
- Adding entity instances to the `GameLayer`
- Reacting to entity logic in the `GameLayer`

The finished game will look like this:



Introduction to game entities

Game entities are classes that define objects needing rendering, collision, physics, or artificial intelligence logic. Fortunately, the entities present in a game's code base often match the conceptual objects in a game. When this is

true, identifying the entities needed in a game can be more easily accomplished.

For example, a space themed shoot 'em up game may include the following entities:

- `PlayerShip`
- `EnemyShip`
- `PlayerBullet`
- `EnemyBullet`
- `CollectableItem`
- `HUD`
- `Environment`

These entities would be their own classes in the game, and each instance would require little or no setup beyond instantiation.

General vs. specific entity types

One of the first questions faced by game developers using an entity system is how much to generalize their entities. The most specific of implementations would define classes for every type of entity, even if they differ by few characteristics. More general systems will combine groups of entities into one class, and allow instances to be customized.

For example we can imagine a space game that defines the following classes:

- `PlayerDogfighter`
- `PlayerBomber`
- `EnemyMissileShip`
- `EnemyLaserShip`

A more-generalized approach would be to create a single class for player ships and a single class for enemy ships, which could be configured to support different ship types. Customization may include which image to load, which type of bullet entities to create when shooting, movement coefficients, and AI logic for the enemy ships. In this case the list of entities may be reduced to:

- `PlayerShip`
- `EnemyShip`

Of course, these entity types can be further generalized by allowing per-instance customization for controlling movement. Player ship instances would read from input while enemy ship instances may perform AI logic. This means that the entities could be generalized even further into a single class:

- `Ship`

The generalization can continue even further – a game may use a single base class for all entities. This single class, which may be called `GameEntity`, would be the class used for every entity instance in the entire game, including entities that are not ships such as bullets and collectible items (power-ups).

This most general of systems is often referred to as a *component system*. In such a system, game entities can have individual components such as physics, AI, or rendering components added to customize behavior and appearance. Pure component-based systems enable ultimate flexibility and can avoid issues caused by the use of inheritance such as complex inheritance chains. As with other generalizations, effective component systems can be difficult to set up and can reduce the expressiveness of code.

The level of generalization used depends on many considerations, including:

- Game size – smaller games can afford to create specific classes, while larger games may be difficult to manage

with a large number of classes.

- Data driven development – games that rely on data (images, 3D models, and data files such as JSON or XML) may benefit from having generalized entity types, and configuring the specifics based on data. This is especially import for games that expect to add new content during development or after the game has been released.
- Game engine patterns – some game engines strongly encourage the usage of component systems while others allow developers to decide how to organize entities. CocosSharp does not require the usage of a component system, so developers are free to implement any type of entity.

For the sake of simplicity, we'll be using a specific class-based approach with a single ship and bullet entity for this tutorial.

Project setup

Before we begin implementing our entities, we need to create a project. We'll be using the CocosSharp project templates to simplify project creation. Check this post for information on creating a CocosSharp project from the Visual Studio for Mac templates. The remainder of this guide will use the project name **EntityProject**.

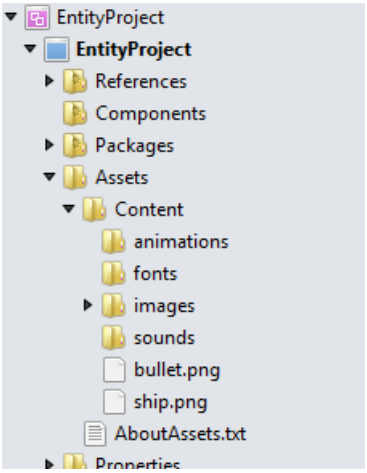
Once our project is created we'll set the resolution of our game to run at 480x320. To do this, call `CCScene.SetDefaultDesignResolution` in the `GameAppDelegate.ApplicationDidFinishLaunching` method as follows:

```
public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
    ...

    // New code for resolution setting:
    CCScene.SetDefaultDesignResolution(480, 320, CCSceneResolutionPolicy.ShowAll);

    CCScene scene = new CCScene (mainWindow);
    GameLayer gameLayer = new GameLayer ();

    scene.AddChild (gameLayer);
    mainWindow.RunWithScene (scene);
}
```

For more information on dealing with CocosSharp resolutions, see our guide on Handling Multiple Resolutions in CocosSharp.

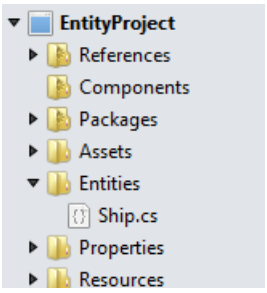
Adding content to the project

Once our project has been created, we will add the files contained in this content zip file. To do this, download the zip file and unzip it. Add both **ship.png** and **bullet.png** to the **Content** folder. The **Content** folder will be inside the **Assets** folder on Android, and will be at the root of the project on iOS. Once added, we should see both files in the **Content** folder:



Creating the ship entity

The `Ship` class will be our game's first entity. To add a `Ship` class, first create a folder called **Entities** at the root level of the project. Add a new class in the **Entities** folder called `Ship`:



The first change we'll make to our `Ship` class is to let it inherit from the `CCNode` class. `CCNode` serves as the base class for common CocosSharp classes like `CCSprite` and `CCLayer`, and gives us the following functionality:

- `Position` property for moving the ship around the screen.
- `Children` property for adding a `CCSprite.`
- `Parent` property, which can be used to attach `Ship` instances to other `CCNodes`. We won't be using this feature in this tutorial; larger games often take advantage of attaching entities to other `CCNodes`.
- `AddEventListener` method for responding to input for moving the ship.
- `Schedule` method for shooting bullets.

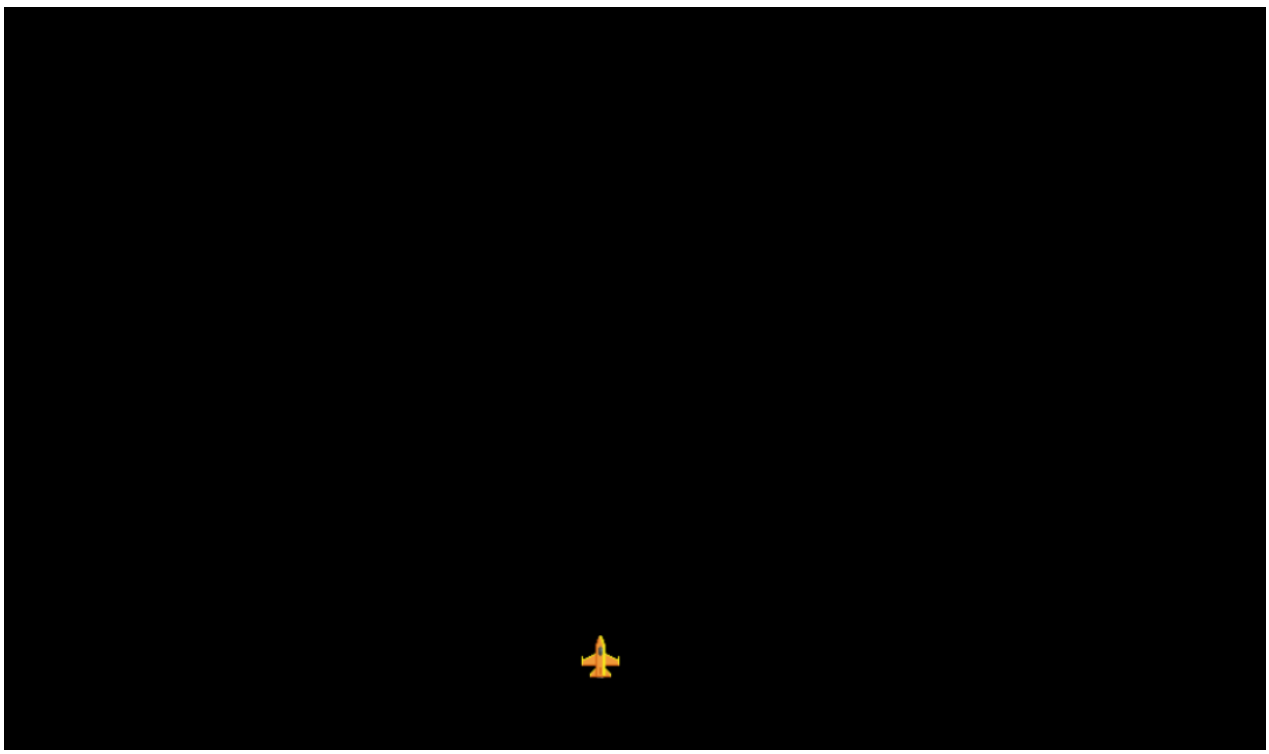
We'll also add a `CCSprite` instance so that our ship can be seen on screen:

```
using System;
using CocosSharp;

namespace EntityProject
{
    public class Ship : CCNode
    {
        CCSprite sprite;

        public Ship () : base()
        {
            sprite = new CCSprite ("ship.png");
            // Center the Sprite in this entity to simplify
            // centering the Ship when it is instantiated
            sprite.AnchorPoint = CCPoint.AnchorMiddle;
            this.AddChild(sprite);
        }
    }
}
```

Next, We'll add the ship to our `GameLayer` to see it show up in our game:

```
public class GameLayer : CCLayer
{
    Ship ship;

    public GameLayer ()
    {
        ship = new Ship ();
        ship.PositionX = 240;
        ship.PositionY = 50;
        this.AddChild (ship);
    }
  ...
```

If we run our game we will now see our Ship entity:



Why inherit from CCNode instead of CCSprite?

At this point our `Ship` class is a simple wrapper for a `CCSprite` instance. Since `CCSprite` also inherits from `CCNode`, we could have inherited directly from `CCSprite`, which would have reduced the code in `Ship.cs`. Furthermore, inheriting directly from `CCSprite` reduces the number of in-memory objects and can improve performance by making the dependency tree smaller.

Despite these benefits, we inherited from `CCNode` to hide some of the `CCSprite` properties from each instance. For example, the `Texture` property should not be modified outside of the `Ship` class, and inheriting from `CCNode` allows us to hide this property. The public members of our entities become especially important as a game grows larger and as additional developers are added to a team.

Adding input to the ship

Now that our ship is visible on screen we will be adding input. Our approach will be similar to the approach taken in the BouncingGame guide, except that we will be placing the code for movement in the `Ship` class rather than in the containing `CCLayer` or `CCScene`.

Add the code to `Ship` to support moving it to wherever the user is touching the screen:

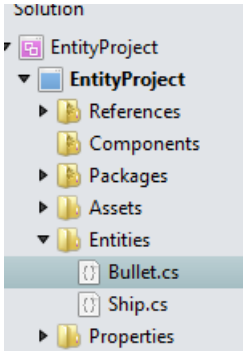
```
public class Ship : CCNode
{
    CCSprite sprite;

    CCEventListenerTouchAllAtOnce touchListener;

    public Ship () : base()
    {
        sprite = new CCSprite ("ship.png");
        // Center the Sprite in this entity to simplify
        // centering the Ship on screen
        sprite.AnchorPoint = CCPoint.AnchorMiddle;
        this.AddChild(sprite);

        touchListener = new CCEventListenerTouchAllAtOnce();
        touchListener.OnTouchesMoved = HandleInput;
        AddEventListener(touchListener, this);

    }

    private void HandleInput(System.Collections.Generic.List<CCTouch> touches, CCEvent touchEvent)
    {
        if(touches.Count > 0)
        {
            CCTouch firstTouch = touches[0];

            this.PositionX = firstTouch.Location.X;
            this.PositionY = firstTouch.Location.Y;
        }
    }
}
```

Many shoot 'em up games implement a maximum velocity, mimicking traditional controller-based movement. That said, we'll simply implement immediate movement to keep our code shorter.

Creating the bullet entity

The second entity in our simple game is an entity for displaying bullets. Just like the `Ship` entity, the `Bullet` entity will contain a `CCSprite` so that it appears on screen. The logic for movement differs in that it does not depend on user input for movement; rather, `Bullet` instances will move in a straight line using velocity properties.

First we'll add a new class file to our **Entities** folder and call it **Bullet**:



Once added we'll modify the `Bullet.cs` code as follows:

```
using System;
using CocosSharp;

namespace EntityProject
{
    public class Bullet : CCNode
    {
        CCSprite sprite;

        public float VelocityX
        {
            get;
            set;
        }

        public float VelocityY
        {
            get;
            set;
        }

        public Bullet () : base()
        {
            sprite = new CCSprite ("bullet.png");
            // Making the Sprite be centered makes
            // positioning easier.
            sprite.AnchorPoint = CCPoint.AnchorMiddle;
            this.AddChild(sprite);

            this.Schedule (ApplyVelocity);
        }

        void ApplyVelocity(float time)
        {
            PositionX += VelocityX * time;
            PositionY += VelocityY * time;
        }
    }
}
```

Aside from changing the file used for the `CCSprite` to `bullet.png`, the code in `ApplyVelocity` includes movement logic which is based on two coefficients: `VelocityX` and `VelocityY`.

The `Schedule` method allows adding delegates to be called every-frame. In this case we're adding the `ApplyVelocity` method so that our bullet moves according to its velocity values. The `Schedule` method takes an `Action<float>`, where the float parameter specifies the amount of time (in seconds) since the last frame, which we use to implement time-based movement. Since the time value is measured in seconds, then our velocity values represent movement in *pixels per second*.

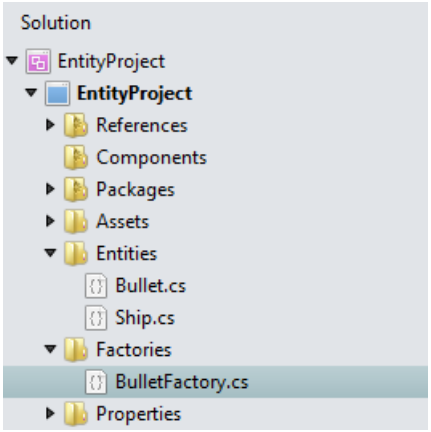
Adding bullets to GameLayer

Before we add any `Bullet` instances to our game we will make a container, specifically a `List<Bullet>`. Modify the `GameLayer` so it includes a list of Bullets:

```
public class GameLayer : CCLayer
{
    Ship ship;
    List<Bullet> bullets;

    public GameLayer ()
    {
        ship = new Ship ();
        ship.PositionX = 240;
        ship.PositionY = 50;
        this.AddChild (ship);

        bullets = new List<Bullet> ();
    }
    ...
```

Next we'll need to populate the `Bullet` list. The logic for when to create a `Bullet` should be contained in the `Ship` entity, but the `GameLayer` is responsible for storing the list of Bullets. We will use the factory pattern to allow the `Ship` entity to create `Bullet` instances. This factory will expose an event that the `GameLayer` can handle.

To do this first, we'll add a folder to our project called **Factories**, and then add a new class called `BulletFactory`:



Next, we'll implement the `BulletFactory` singleton class:

```
using System;

namespace EntityProject
{
    public class BulletFactory
    {
        static Lazy<BulletFactory> self =
            new Lazy<BulletFactory>(()=>new BulletFactory());

        // simple singleton implementation
        public static BulletFactory Self
        {
            get
            {
                return self.Value;
            }
        }

        public event Action<Bullet> BulletCreated;

        private BulletFactory()
        {

        }

        public Bullet CreateNew()
        {
            Bullet newBullet = new Bullet ();

            if (BulletCreated != null)
            {
                BulletCreated (newBullet);
            }

            return newBullet;
        }
    }
}
```

The `Ship` entity will handle creating `Bullet` instances – specifically, it will handle how frequently `Bullet` instances should be created (i.e. how often the bullet is fired), their position, and their velocity.

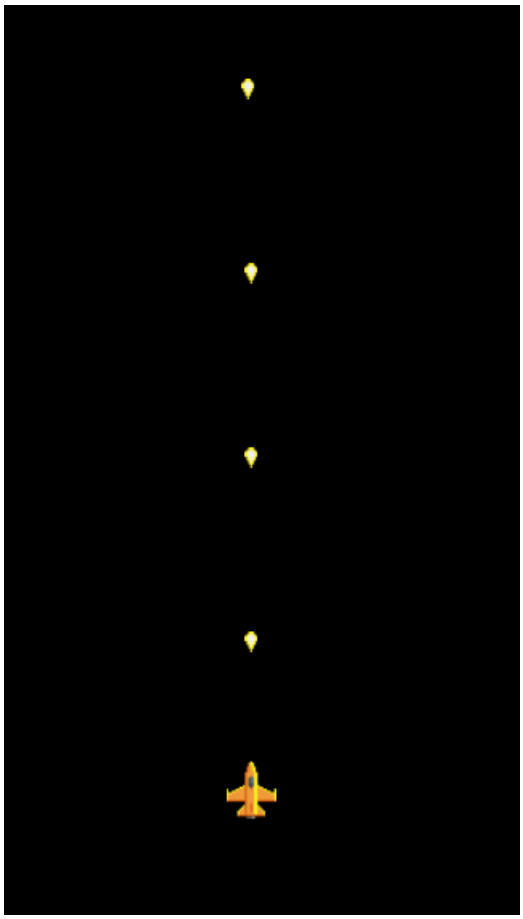
Modify the `Ship` entity's constructor to add a new `Schedule` call, and then implement this method as follows:

```
...
public Ship () : base()
{
    sprite = new CCSprite ("ship.png");
    // Center the Sprite in this entity to simplify
    // centering the Ship on screen
    sprite.AnchorPoint = CCPoint.AnchorMiddle;
    this.AddChild(sprite);

    touchListener = new CCEventListenerTouchAllAtOnce();
    touchListener.OnTouchesMoved = HandleInput;
    AddEventListener(touchListener, this);

    Schedule (FireBullet, interval: 0.5f);

}

void FireBullet(float unusedValue)
{
    Bullet newBullet = BulletFactory.Self.CreateNew ();
    newBullet.Position = this.Position;
    newBullet.VelocityY = 100;
}
...
```

The last step is to handle the creation of new `Bullet` instances in the `GameLayer` code. Add an event handler to the `BulletCreated` event that adds the newly created `Bullet` to the appropriate lists:

```
...
public GameLayer ()
{
    ship = new Ship ();
    ship.PositionX = 240;
    ship.PositionY = 50;
    this.AddChild (ship);

    bullets = new List<Bullet> ();
    BulletFactory.Self.BulletCreated += HandleBulletCreated;
}

void HandleBulletCreated(Bullet newBullet)
{
    AddChild (newBullet);
    bullets.Add (newBullet);
}
...
```

Now we can run the game and see the `Ship` shooting `Bullet` instances:



Why GameLayer has ship and bullets members

Our `GameLayer` class defines two fields to hold references to our entity instances (`ship` and `bullets`), but does nothing with them. Furthermore, entities are responsible for their own behavior such as movement and shooting. So why did we add `ship` and `bullets` fields to `GameLayer`?

The reason we added these members is because a full game implementation would require logic in the `GameLayer` for interaction between the different entities. For example, this game may be further developed to include enemies that can be destroyed by the player. These enemies would be contained in a `List` in the `GameLayer`, and logic to test whether `Bullet` instances collide with the enemies would be performed in the `GameLayer` as well. In other words, the `GameLayer` is the root *owner* of all entity instances, and it is responsible for interactions between entity instances.

Bullet destruction considerations

Our game currently lacks code for destroying `Bullet` instances. Each `Bullet` instance has logic for moving on screen, but we haven't added any code to destroy any off-screen `Bullet` instances.

Furthermore, the destruction of `Bullet` instances may not belong in `GameLayer`. For example, rather than being destroyed when off-screen, the `Bullet` entity may have logic to destroy itself after a certain amount of time. In this case, the `Bullet` needs a way to communicate that it should be destroyed to the `GameLayer`, much like the `Ship` entity communicated to the `GameLayer` that a new `Bullet` was created through the `BulletFactory`.

The simplest solution is to expand the responsibility of the factory class to support destruction. Then the factory can be notified of an entity instance being destroyed, which can be handled by other objects, such as the `GameLayer` removing the entity instance from its lists.

Summary

This guide shows how to create CocosSharp entities by inheriting from the `CCNode` class. These entities are self-contained objects, handling creation of their own visuals and custom logic. This guide designates code that belongs inside an entity (movement and creation of other entities) from code that belongs in the root entity container (collision and other entity interaction logic).

Related links

- CocosSharp API Documentation
- Content zip

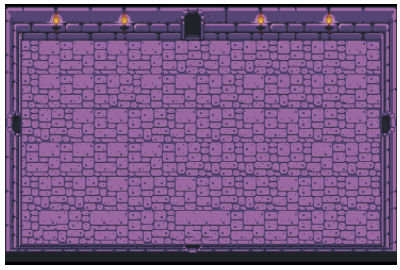
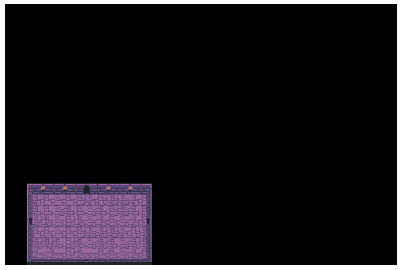
Handling multiple resolutions in CocosSharp

10/3/2018 • 10 minutes to read • Edit Online

This guide shows how to work with CocosSharp to develop games that display properly on devices of varying resolutions.

CocosSharp provides methods for standardizing object dimensions in your game regardless of the physical number of pixels on a device's display. Traditionally, games developed for PCs and gaming consoles could target a single resolution. Modern games – and especially games for mobile devices – must be built to accommodate a wide variety of devices. This guide shows how to standardize CocosSharp games regardless of the resolution characteristics of the physical display.

The default resolution behavior of CocosSharp is to match physical pixels with in-game coordinates. The following table shows how various devices would render a background environment sprite with width and height of 368x240. The first row is technically not an actual device, but rather the expected rendering of the sprite, regardless of device resolution:

DEVICE	DISPLAY RESOLUTION	EXAMPLE SCREENSHOT
Desired Display	368x240 (with black bars for aspect ratio)	
iPhone 4s	960x640	
iPhone 6 Plus	1920x1080	

This document covers how to use CocosSharp to fix the problem shown in the table above. That is, we'll cover how to make any device render as shown in the first row – regardless of screen resolution.

Working with SetDesignResolutionSize

The `CCScene` class is typically used as the root container for all visual objects, but it also provides a static method called `SetDesignResolutionSize` for specifying the default size for all scenes. In other words `SetDesignResolutionSize` method allows developers to develop games which will display correctly on a variety of

hardware resolutions. The CocosSharp project templates use this method to set the default project size to 1024x768, as is shown in the following code:

```
public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
 application.PreferMultiSampling = false;
 application.ContentRootDirectory = "Content";
 application.ContentSearchPaths.Add ("animations");
 application.ContentSearchPaths.Add ("fonts");
 application.ContentSearchPaths.Add ("sounds");
 CCSize windowSize = mainWindow.WindowSizeInPixels;
 float desiredWidth = 1024.0f;
 float desiredHeight = 768.0f;

 // This will set the world bounds to (0,0, w, h)
 // CCSceneResolutionPolicy.ShowAll will ensure that the aspect ratio is preserved
 CCScene.SetDesignResolutionSize (desiredWidth, desiredHeight, CCSceneResolutionPolicy.ShowAll);
 ...
```

You can change the `desiredWidth` and `desiredHeight` variables above to modify the display to match the desired resolution of your game. For example, the above code could be modified as follows to display the 368x240 background as full-screen:

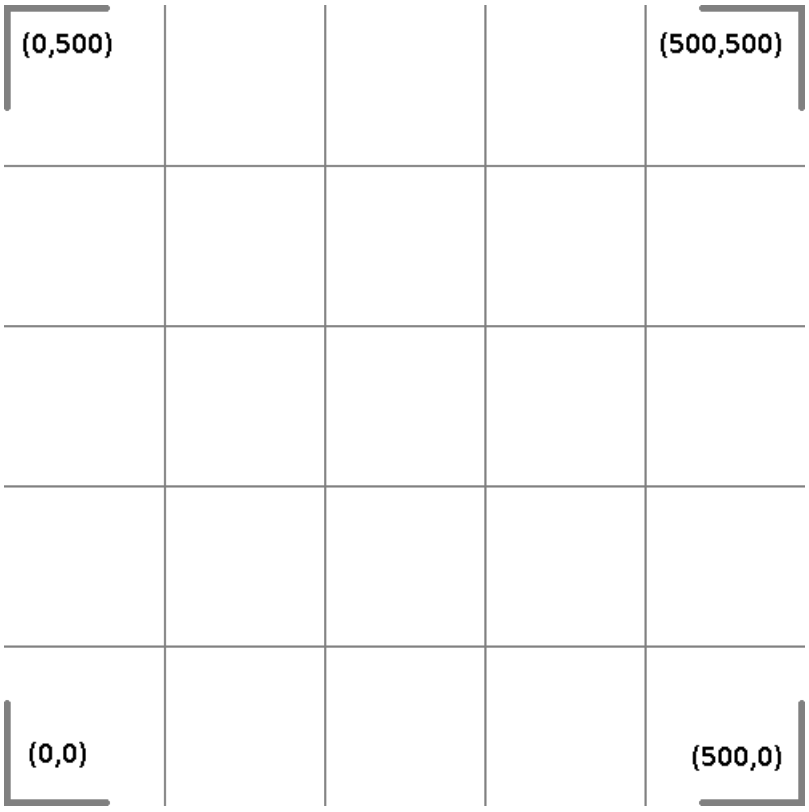
```
public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
 application.PreferMultiSampling = false;
 application.ContentRootDirectory = "Content";
 application.ContentSearchPaths.Add ("animations");
 application.ContentSearchPaths.Add ("fonts");
 application.ContentSearchPaths.Add ("sounds");
 CCSize windowSize = mainWindow.WindowSizeInPixels;
 float desiredWidth = 368.0f;
 float desiredHeight = 240.0f;

 // This will set the world bounds to(0,0, w, h)
 // CCSceneResolutionPolicy.ShowAll will ensure that the aspect ratio is preserved
 CCScene.SetDesignResolutionSize (desiredWidth, desiredHeight, CCSceneResolutionPolicy.ShowAll);
 ...
```

CCSceneResolutionPolicy

`SetDesignResolutionSize` allows us to specify how the game window adjusts to the desired resolution. The following sections demonstrate how a 500x500 image is displayed with different `CCSceneResolutonPolicy` values passed to the `SetDesignResolutionSize` method. The following values are provided by the `CCSceneResolutionPolicy` enum:

- `ShowAll` – Displays the entire requested resolution, maintaining aspect ratio.
- `ExactFit` – Displays the entire requested resolution, but does not maintain aspect ratio.
- `FixedWidth` – Displays the entire requested width, maintaining aspect ratio.
- `FixedHeight` – Displays the entire requested height, maintaining aspect ratio.
- `NoBorder` – Displays either the entire height or entire width, maintaining the aspect ratio. If the aspect ratio of the device is greater than the aspect ratio of the desired resolution, then the entire width is shown. If the aspect ratio of the device is less than the aspect ratio of the desired resolution, then the entire height is shown.
- `Custom` – The display depends on the `Viewport` property of the current `CCScene`.

All screenshots are produced at iPhone 4s resolution (960x640) in landscape orientation and use this image:



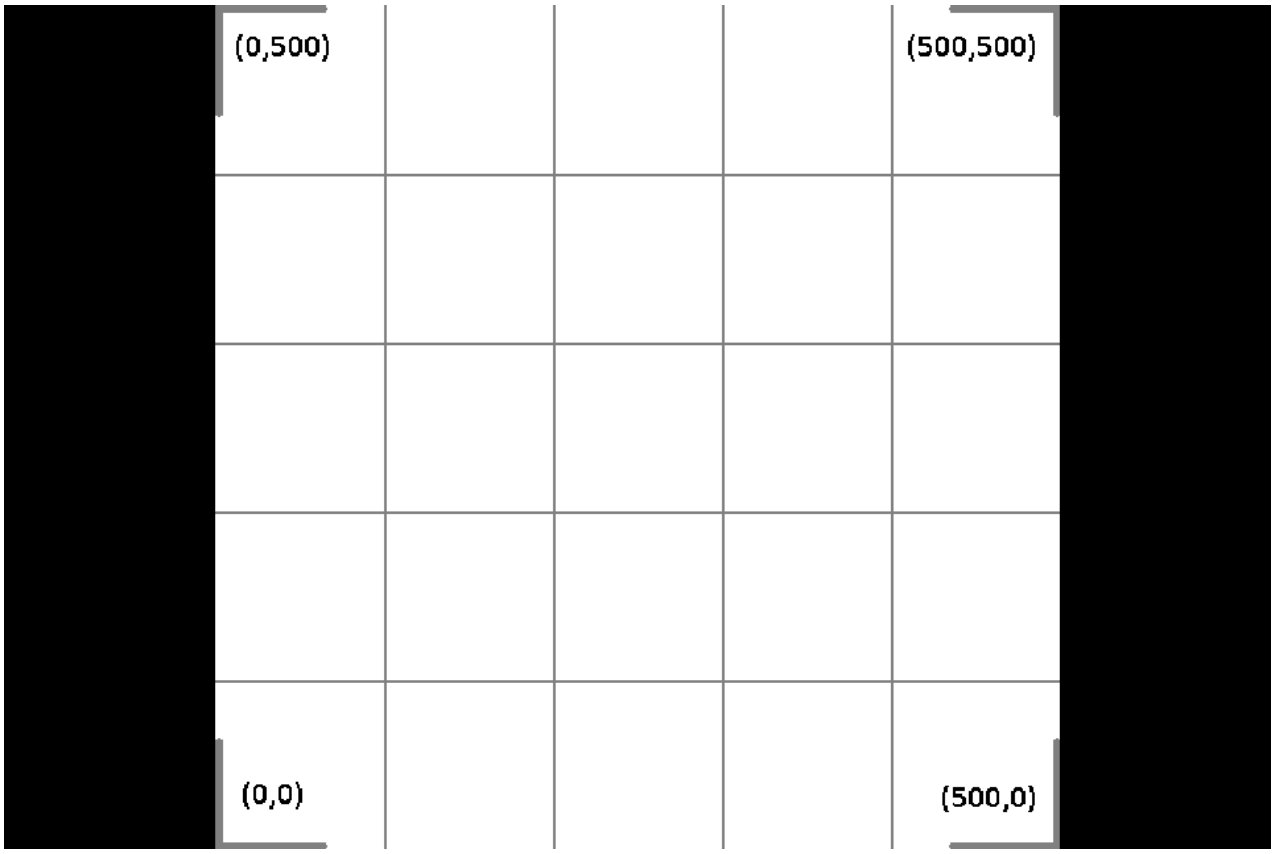
CCSceneResolutionPolicy.ShowAll

`ShowAll` specifies that the entire game resolution will be visible on-screen, but may display *letterboxing* (black bars) to adjust for different aspect ratios. This policy is commonly used as it guarantees the entire game view will be displayed on-screen without any distortion.

Code example:

```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.ShowAll);
```

Letterboxing is visible to the left and right of the image to account for the physical aspect ratio being wider than the desired resolution:



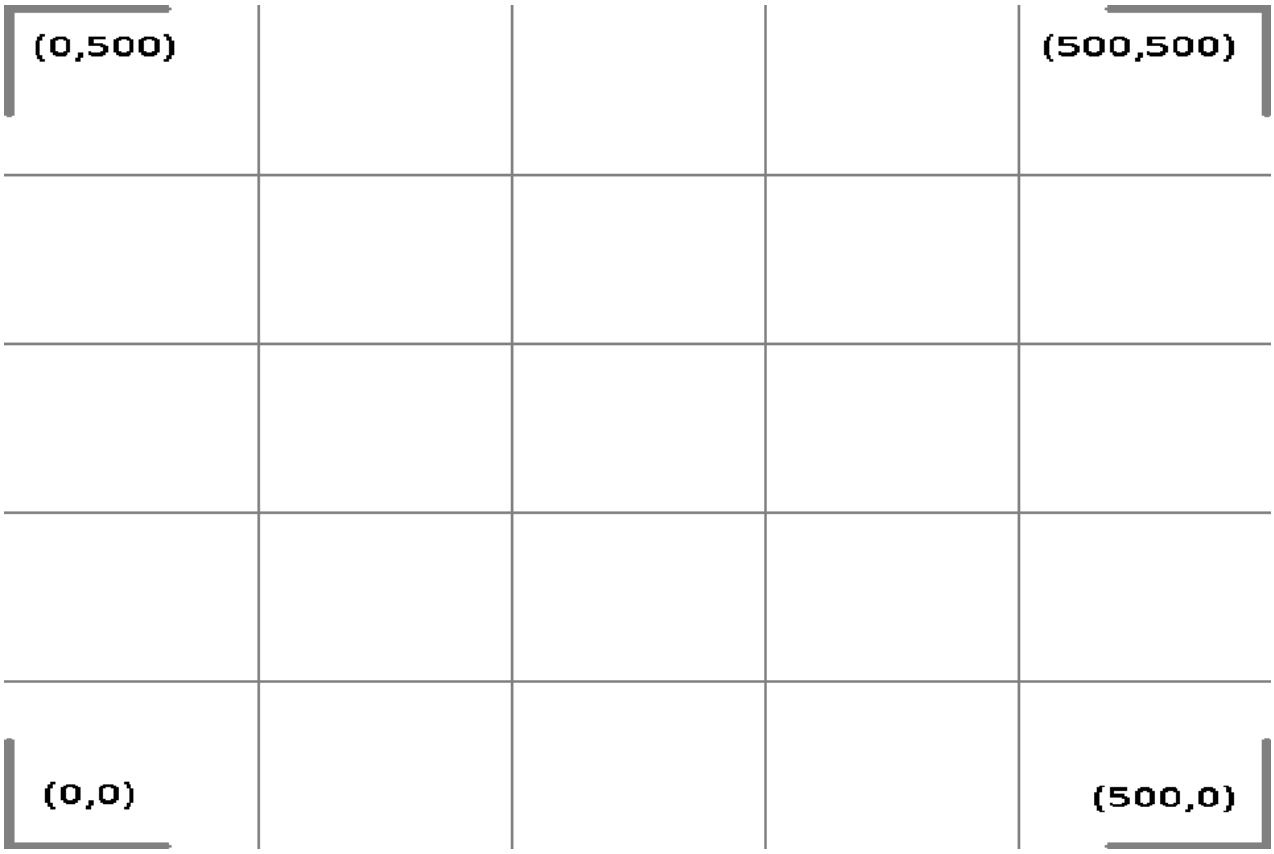
CCSceneResolutionPolicy.ExactFit

`ExactFit` specifies that the entire game resolution will be visible on-screen with no letterboxing. The viewable area may be distorted (aspect ratio may not be maintained) according to hardware aspect ratio.

Code example:

```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.ExactFit);
```

No letterboxing is visible, but since the device resolution is rectangular the game view is distorted:



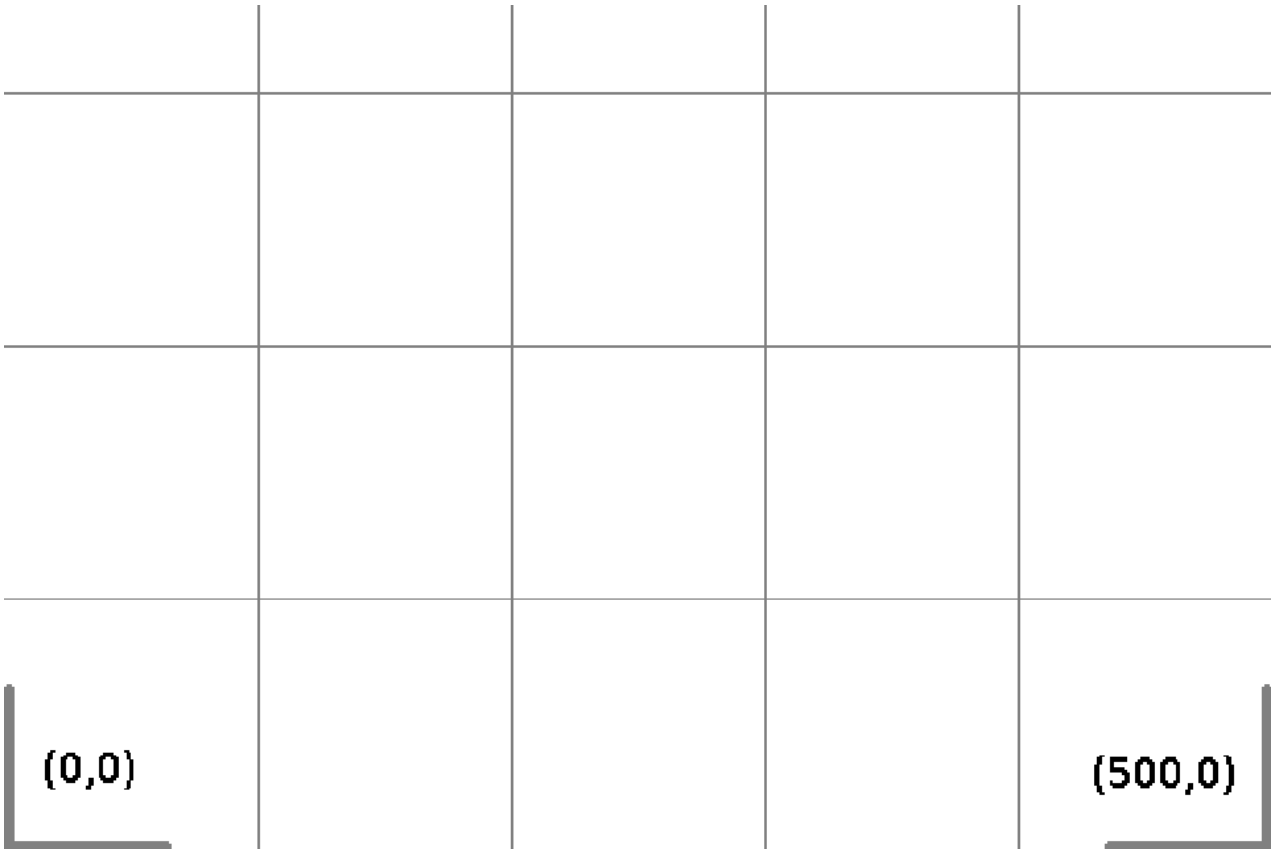
CCSceneResolutionPolicy.FixedWidth

`FixedWidth` specifies that the width of the view will match the width value passed to `SetDesignResolutionSize`, but the viewable height is subject to the aspect ratio of the physical device. The height value passed to `SetDesignResolutionSize` is ignored since it will be calculated at runtime based on the physical device's aspect ratio. This means that the calculated height may be smaller than the desired height (which results in parts of the game view being off-screen), or the calculated height may be larger than the desired height (which results in more of the game view being displayed). Since this may result in more of the game being displayed, then it may appear as if letterboxing has occurred; however, the extra space will not necessarily be black if any visual object appears there.

Code example:

```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.FixedWidth);
```

The iPhone 4s has an aspect ratio of 3:2, so the calculated height is approximately 333 units:



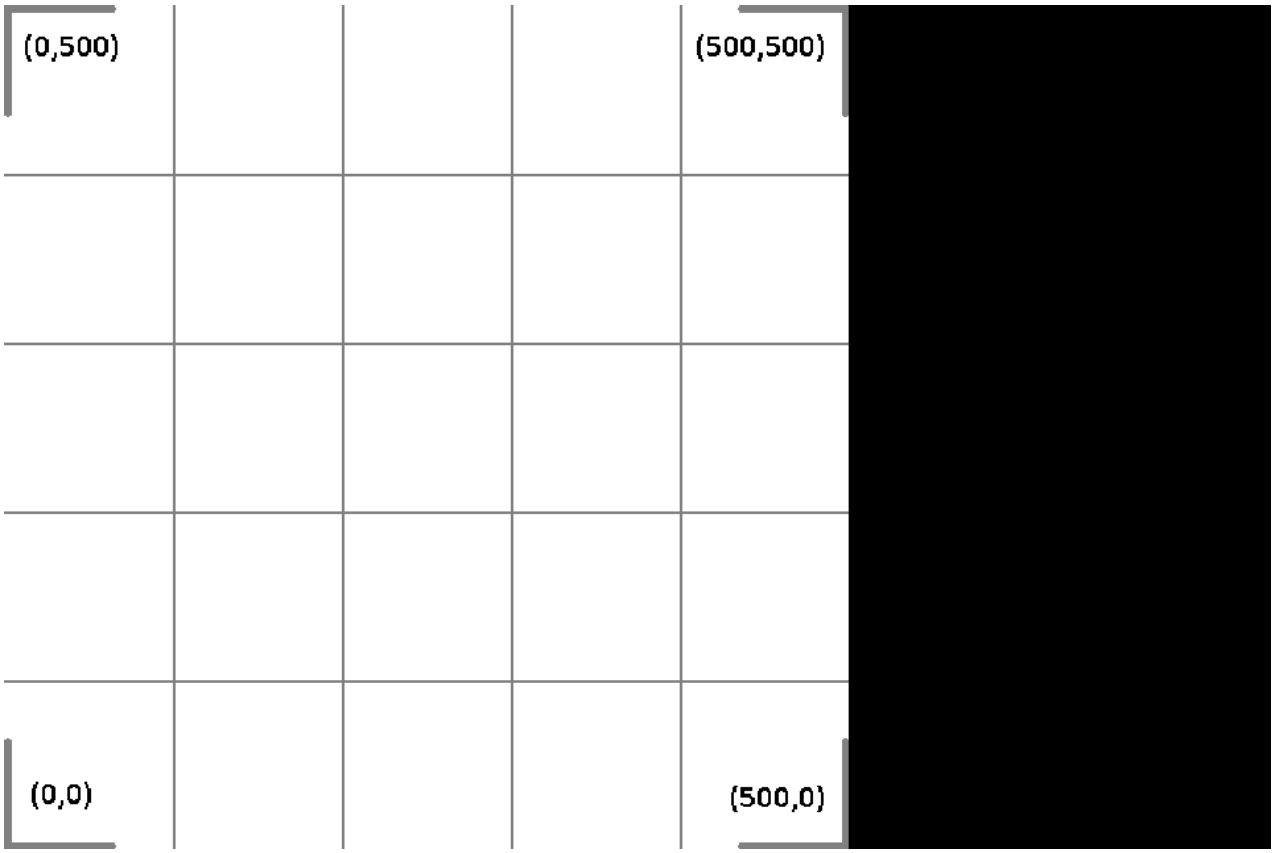
CCSceneResolutionPolicy.FixedHeight

Conceptually, `FixedHeight` behaves similarly to `FixedWidth` – the game will obey the height value passed to `SetDesignResolutionSize,` but will calculate the width at runtime based off of the physical resolution. As mentioned above, this means that the displayed width be smaller or larger than the desired width, resulting in part of the game being off screen or more of the game being displayed, respectively.

Code example:

```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.FixedHeight);
```

Since the following screenshot was created with the app running in landscape mode, the calculated width is larger than 500 – specifically 750. This policy keeps the X value of 0 left-aligned, so the extra resolution is viewable on the right side of the screen.



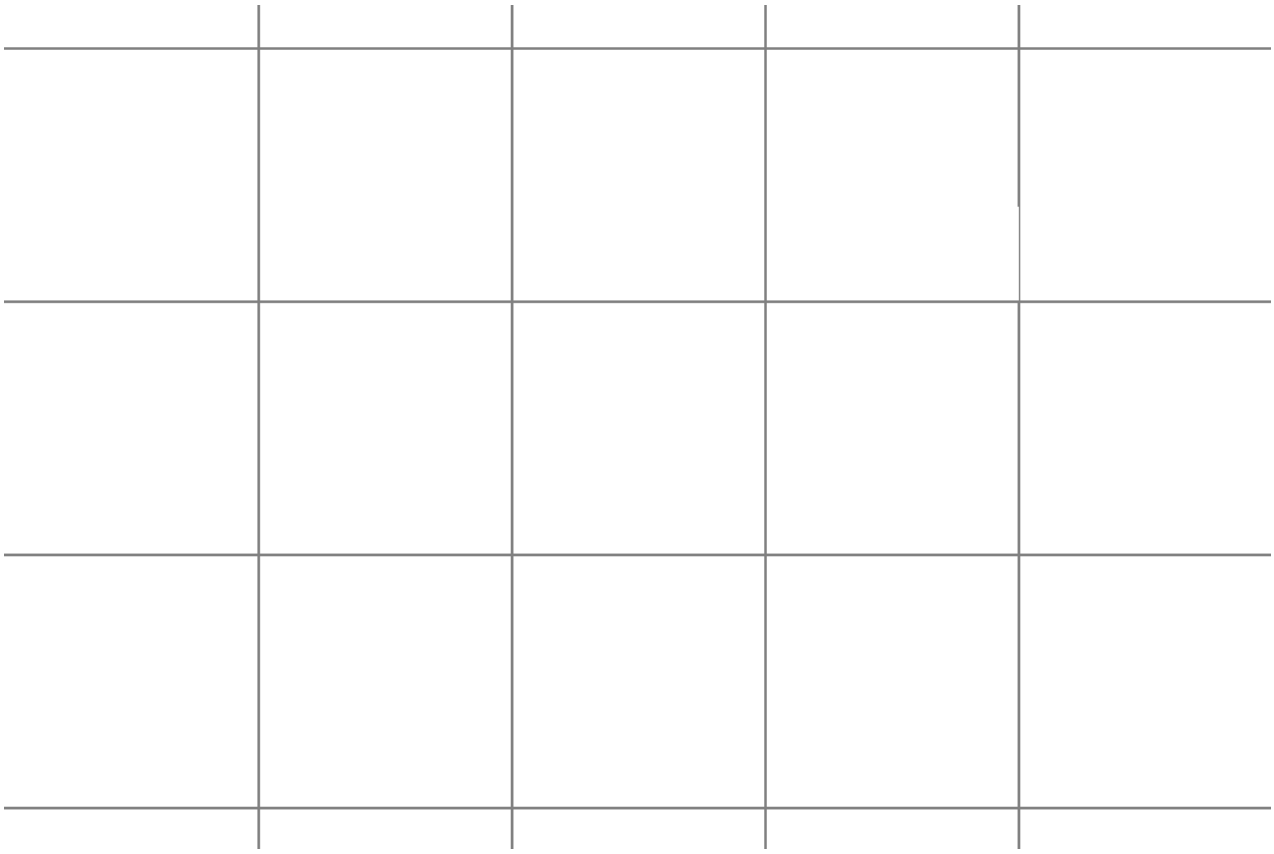
CCSceneResolutionPolicy.NoBorder

`NoBorder` attempts to display the application with no letterboxing while maintaining the original aspect ratio (no distortion). If the requested resolution's aspect ratio matches the device's physical aspect ratio, then no clipping will occur. If aspect ratios do not match, then clipping will occur.

Code example:

```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.FixedHeight);
```

The following screen shot displays the top and bottom parts of the display clipped, while all 500 pixels of the display width are displayed:



CCSceneResolutionPolicy.Custom

`Custom` enables each `CCScene` to specify its own custom viewport relative to the resolution specified in `SetDesignResolutionSize`.

Scenes define their `Viewport` property by constructing a `CCViewport`, which in turn is constructed with a `CCRect`. For more information on `CCViewport` and `CCRect` see the CocosSharp API Documentation. In the context of creating a `CCViewport` the `CCRect` constructor's parameters specify (in order):

- x – The amount to horizontally offset the viewport, where each unit represents one entire screen width. For example, using a value of .5f results in the scene shifted to the right by half of the screen width.
- y – The amount to vertically offset the viewport, where each unit represents one entire screen height. For example, using a value of .5f results in the scene shifted down by half of the screen height.
- width – The horizontal portion that the scene should occupy. For example, using a value of 1/3.0f results in the scene horizontally occupying one-third of the screen.
- height – The vertical portion that the scene should occupy. For example, using a value of 1/3.0f results in the scene vertically occupying one-third of the screen.

Code example:

```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.Custom);
...
float horizontalDisplayPortion = 2 / 3.0f;
float verticalDisplayPortion = 1 / 2.0f;
float displayOffsetInScreenWidths = 0;
float displayOffsetInScreenHeights = .5f;
var rectangle = new CCRect (
 displayOffsetInScreenWidths,
 displayOffsetInScreenHeights,
 horizontalDisplayPortion,
 verticalDisplayPortion);
scene.Viewport = new CCViewport (rectangle);
```

The code above results in the following:

DefaultTexelToContentSizeRatio

The `DefaultTexelToContentSizeRatio` simplifies using higher-resolution textures on devices with higher resolution screens. Specifically, this property allows games to use higher-resolution assets without needing to change the size or positioning of visual elements.

For example, a game may include an art asset for a game character which is 200 pixels tall, and the game screen (as specified by `SetDesignResolutionSize`) may be 400 pixels tall. In this case, the character will occupy half of the screen. However, if a 400 pixel-tall asset were used for the character for higher-resolution devices, the character would occupy the entire height of the display. If the intent is to keep the character the same size to take advantage of higher resolution devices, then some extra code is necessary to keep the character sprite at half the height of the screen.

The simple example presented above represents a common problem in game development – adding larger-sized assets without requiring the developer to perform resizing on each visual element according to device resolution. `DefaultTexelToContentSizeRatio` is a global property used for resizing visual elements. This value resizes all visual elements of a certain type by the assigned value.

This property is present in the following classes:

- `CCApplication`
- `CCSprite`
- `CCLabelTtf`
- `CCLabelBMFont`
- `CCTMXLayer`

The CocosSharp Visual Studio for Mac templates set `CCSprite.DefaultTexelToContentSizeRatio` according to the width of the device as an example of how it can be used. The following code is contained in `CCApplicationDelegate.ApplicationDidFinishLaunching`:

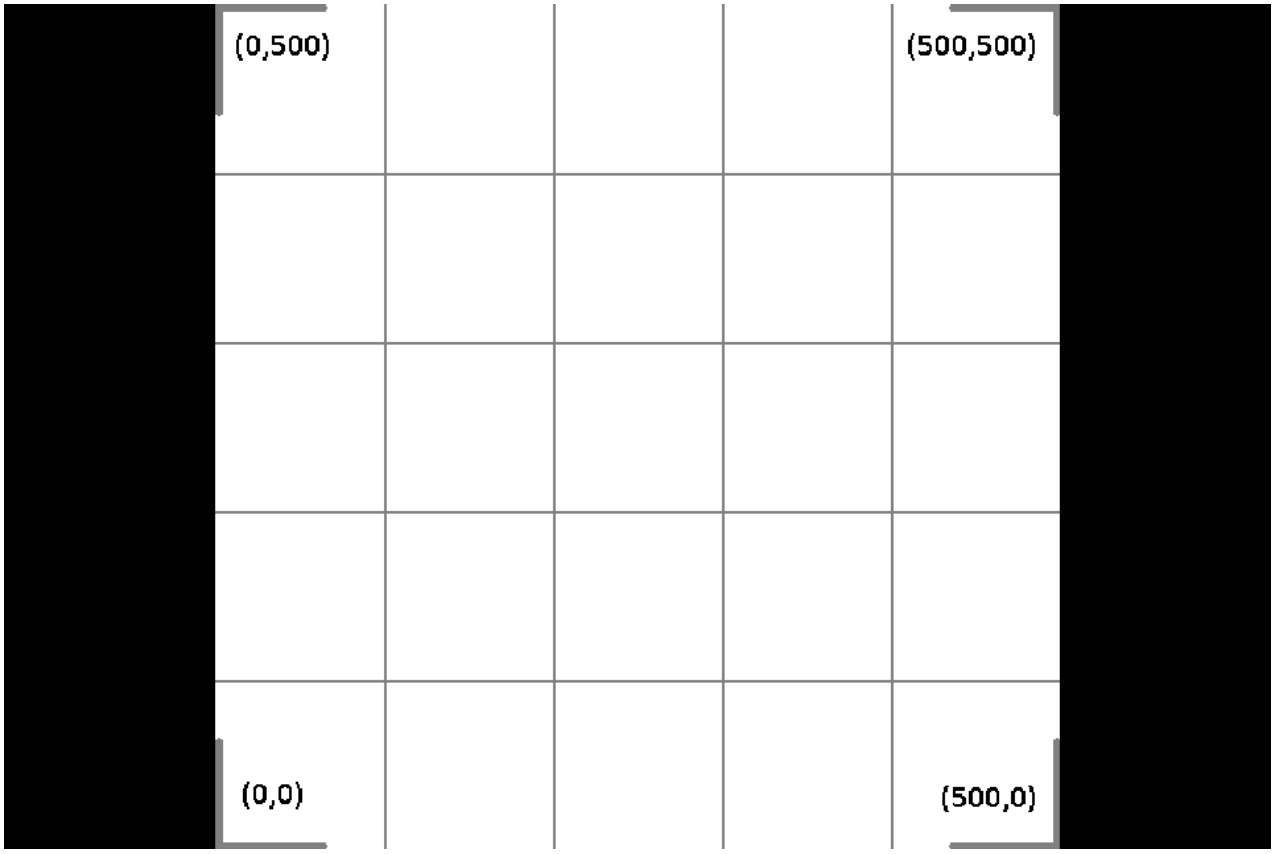
```
public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
 ...
 float desiredWidth = 1024.0f;
 float desiredHeight = 768.0f;

 ...
 if (desiredWidth < windowSize.Width)
 {
  application.ContentSearchPaths.Add ("images/hd");
  CCSprite.DefaultTexelToContentSizeRatio = 2.0f;
 }
 else
 {
  application.ContentSearchPaths.Add ("images/ld");
  CCSprite.DefaultTexelToContentSizeRatio = 1.0f;
 }
 ...
}
```

DefaultTexelToContentSizeRatio example

To see how `DefaultTexelToContentSizeRatio` impacts the size of visual elements, consider the code presented above:

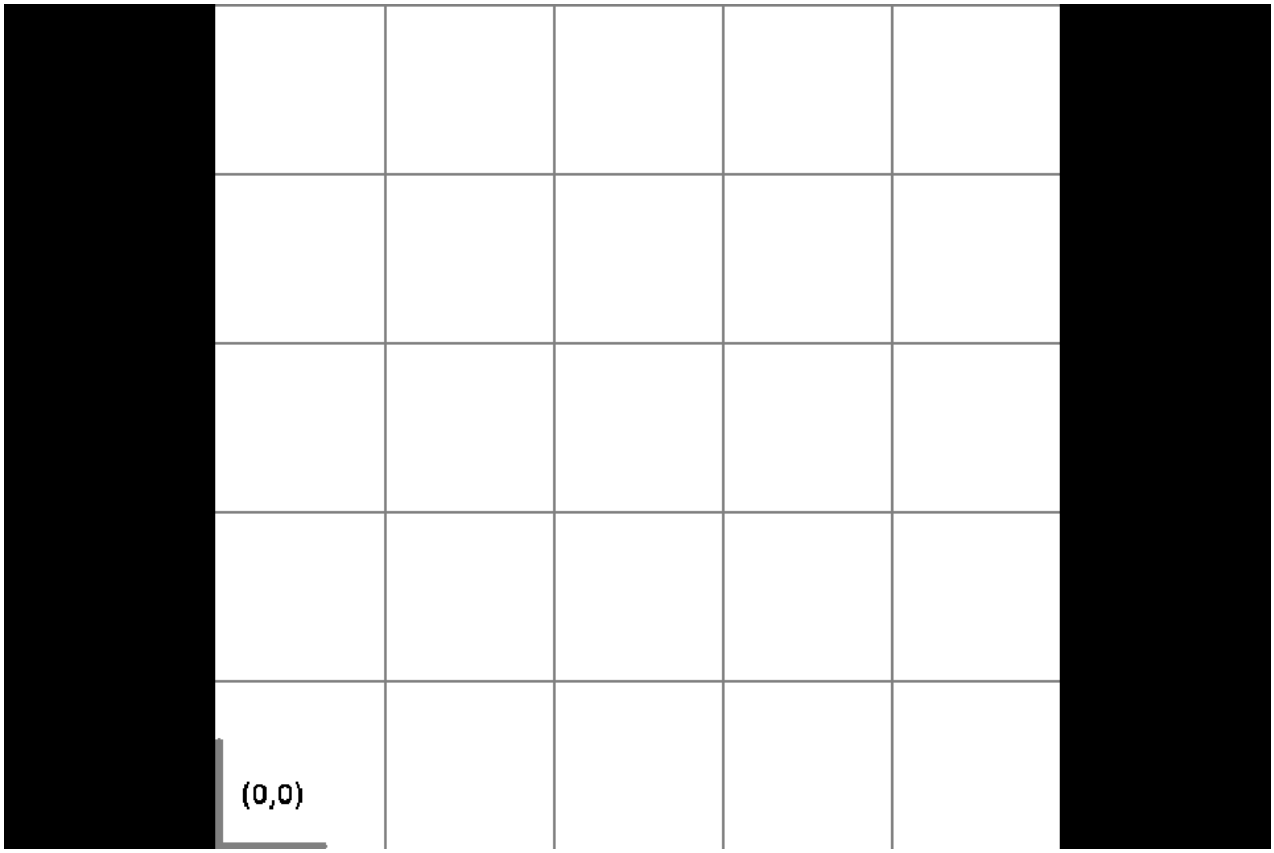
```
CCScene.SetDesignResolutionSize (500.0f, 500.0f, CCSceneResolutionPolicy.ShowAll);
```

Which will result in the following image using a 500x500 texture:



The iPad Retina runs at a physical resolution of 2048x1536. This means that the game as displayed above would stretch 500 pixels over 1536 physical pixels. Games can take advantage of this extra resolution by creating what are typically referred to as *hd* assets – assets which are designed to run on higher resolution screens. For example, this game could use an hd version of this texture sized at 1000x1000. However, using the larger image would result in the `CCSprite` having a width and height of 1000 units. Since our game will always display 500 units (due

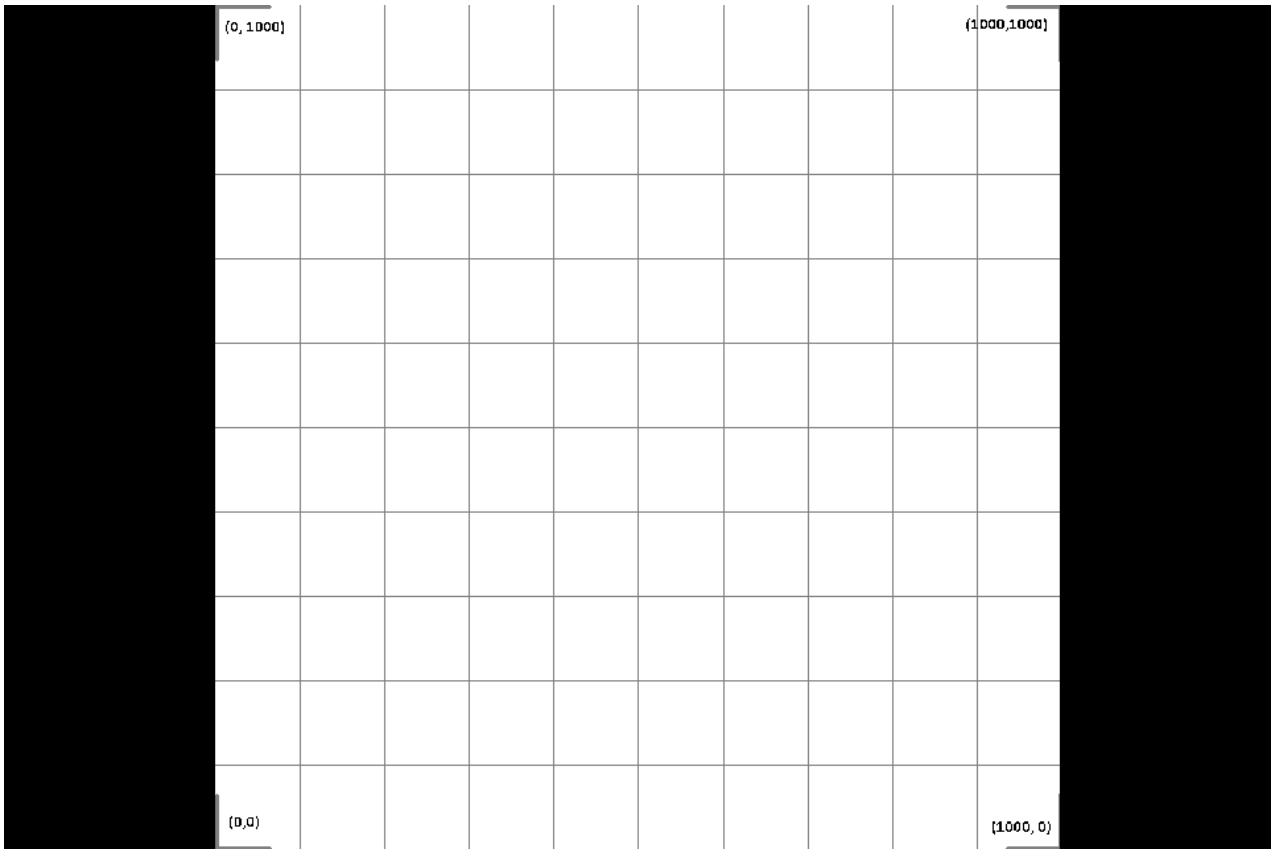
to the `SetDesignResolutioSize` call), then our 1000x1000 sprite would be too big (only the bottom left portion of it displays):



We can set `CCSprite.DefaultTexelToContentSizeRatio` to account for the 1000x1000 texture being twice as wide and tall as the original 500x500 texture:

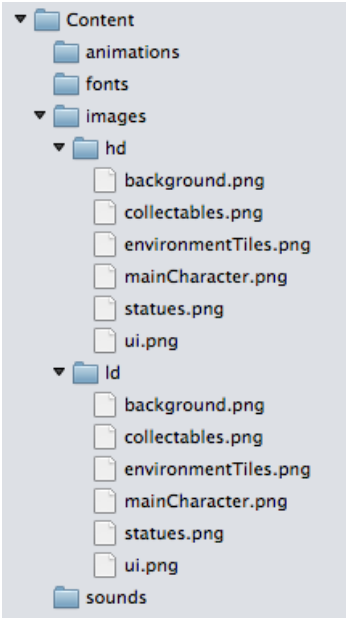
```
CCSprite.DefaultTexelToContentSizeRatio = 2;
```

Now if we run the game the 1000x1000 texture will be fully visible:



DefaultTexelToContentSizeRatio details

The `DefaultTexelToContentSizeRatio` property is `static,` which means all sprites in the application will share the same value. The typical approach for games with assets made for different resolutions is to contain a complete set of assets for each resolution category. By default the CocosSharp Visual Studio for Mac templates provide **ld** and **hd** folders for assets, which would be useful for games supporting two sets of textures. An example content folder with content may look like:



Notice that the default template also includes code to conditionally add to the application's `ContentSearchPaths`:

```
public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
 ...
 if (desiredWidth < windowSize.Width)
 {
  application.ContentSearchPaths.Add ("images/hd");
  CCSprite.DefaultTexelToContentSizeRatio = 2.0f;
 }
 else
 {
  application.ContentSearchPaths.Add ("images/ld");
  CCSprite.DefaultTexelToContentSizeRatio = 1.0f;
 }
 ...
}
```

This means that the application will search for files in paths according to whether it is running in high resolution or regular resolution mode. For example, if the **hd** and **ld** folders contain a file called **background.png** then the following code would run and select the correct file for the resolution:

```
backgroundSprite  = new CCSprite ("background");
```

Summary

This article covers how to create games which display correctly regardless of device resolution. It shows examples of using different `CCSceneResolutionPolicy` values for resizing the game according to the device resolution. It also provides an example of how `DefaultTexelToContentSizeRatio` can be used to accommodate multiple sets of content without requiring visual elements to be resized individually.

Related links

- CocosSharp Introduction
- CocosSharp API Documentation

CocosSharp Content Pipeline

11/11/2018 • 2 minutes to read • Edit Online

Content pipelines are often used in game development to optimize content and format it such that it can be loaded on certain hardware or with certain game development frameworks.

Introduction to content pipelines

This article provides a conceptual understanding of content pipelines, primarily focusing on the MonoGame Content Pipeline, which is a content pipeline implementation used with CocosSharp and MonoGame.

Walkthrough : Using the Content Pipeline with CocosSharp

This walkthrough will useuses a CocosSharp project to demonstrate how .xnb files can be loaded and used in an application. Users of MonoGame will also be able to reference this walkthrough as CocosSharp and MonoGame both use the same .xnb content files.

Introduction to content pipelines

10/3/2018 • 3 minutes to read • Edit Online

Content pipelines are applications, or parts of applications, that are used to convert files into a format that can be loaded by game projects. The MonoGame Content Pipeline is a specific content pipeline implementation for converting files for CocosSharp and MonoGame projects.

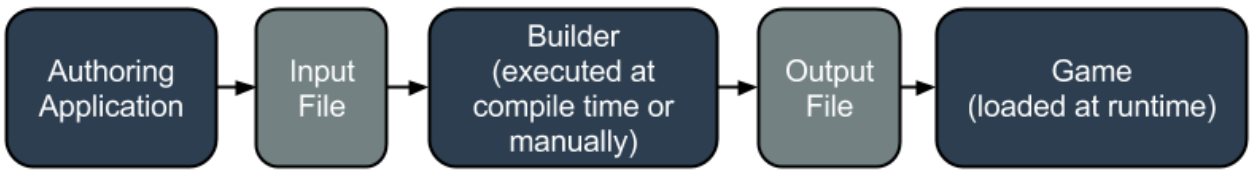
This article provides a conceptual understanding of content pipelines, primarily focusing on the *MonoGame Content Pipeline*, which is a content pipeline implementation used with CocosSharp and MonoGame.

What is a content pipeline?

The term *content pipeline* is a general term for the process of converting a file from one format to another. The *input* of the content pipeline is typically a file outputted by an authoring tool, such as image files from Photoshop. The content pipeline creates the *output* file in a format that can be loaded directly by a game project. Typically the output files are optimized for fast loading and reduced disk size.

Content pipelines may be command-line executables, dedicated GUI-based applications, or plugins embedded in another application such as Visual Studio. Content pipelines typically run before the game executes. If the content pipeline is associated with some application like Visual Studio, then it may execute during compile-time. If the content pipeline is a standalone application, then it may run when explicitly told to do so by the user. The application or logic responsible for converting a specific input file (such as a **.png**) to an associated output file is referred to as a *builder*.

We can visualize the path that a file takes from authoring to being loaded at runtime as follows:



Why use a content pipeline?

Content pipelines introduce an extra step between the authoring application and the game, which can increase compile times and add complexity to the development process. Despite these considerations, content pipelines introduce a number of benefits to game development:

Converting to a format understood by the game

CocosSharp and MonoGame provide methods for loading various types of content; however, the content must be formatted correctly before being loaded. Most types of content require some type of conversion before being loaded. For example, sound effects in the **.wav** format must be converted into an **.xnb** file to be loaded at runtime since CocosSharp and MonoGame do not support loading the **.wav** file format.

Converting to a format native to the hardware

Different hardware may treat content differently at runtime. For example, CocosSharp games can load image files when creating a `CCSprite` instance. Although the same code can be used to load the files on both iOS and Android, each platform stores the loaded file differently. As a consequence, the MonoGame Content Pipeline formats texture **.xnb** files differently depending on the target platform.

Reducing size on disk

Content pipelines can be used to remove information, which is useful at author time but not necessary at runtime. The original (input) file can store all information which can help content creators maintain existing content, but the output file can be stripped-down to keep the overall game file small. This consideration is especially useful for mobile games that are downloaded rather than distributed on installation media.

Reducing load time

Games may require modifications of content to improve runtime performance, to improve visuals, or to add new features. For example many 3D games calculate lighting one time, then use the result of this calculation when rendering complex scenes. Since performing these calculations when loading content can be prohibitively expensive the calculation can instead be performed when the game is built. The resulting calculations can be included in the content, enabling the content to be loaded much faster than would otherwise be possible.

xnb file extension

The **.xnb** file extension is the extension for all files outputted by the Monogame Content Pipeline. This matches the extension of files outputted by Microsoft XNA's Content Pipeline.

The **.xnb** extension is used regardless of the original file type. In other words, image files (**.png**), audio files (**.wav**), and any custom file types will all be outputted as **.xnb** files when passed through the content pipeline. Since the extension cannot be used to distinguish between different file formats then both CocosSharp and MonoGame methods which load **.xnb** files do not expect extensions when loading the file.

CocosSharp and MonoGame .xnb files can be created using the Monogame Pipeline tool which is covered in this walkthrough.

Summary

This article provided an overview and benefits of content pipelines in general, as well as an introduction to the MonoGame Content Pipeline.

Related links

- MonoGame Pipeline Documentation

Using the MonoGame Pipeline Tool

11/5/2018 • 7 minutes to read • Edit Online

The MonoGame Pipeline Tool is used to create and manage MonoGame content projects. The files in content projects are processed by the MonoGame Pipeline tool and outputted as .xnb files for use in CocosSharp and MonoGame applications.

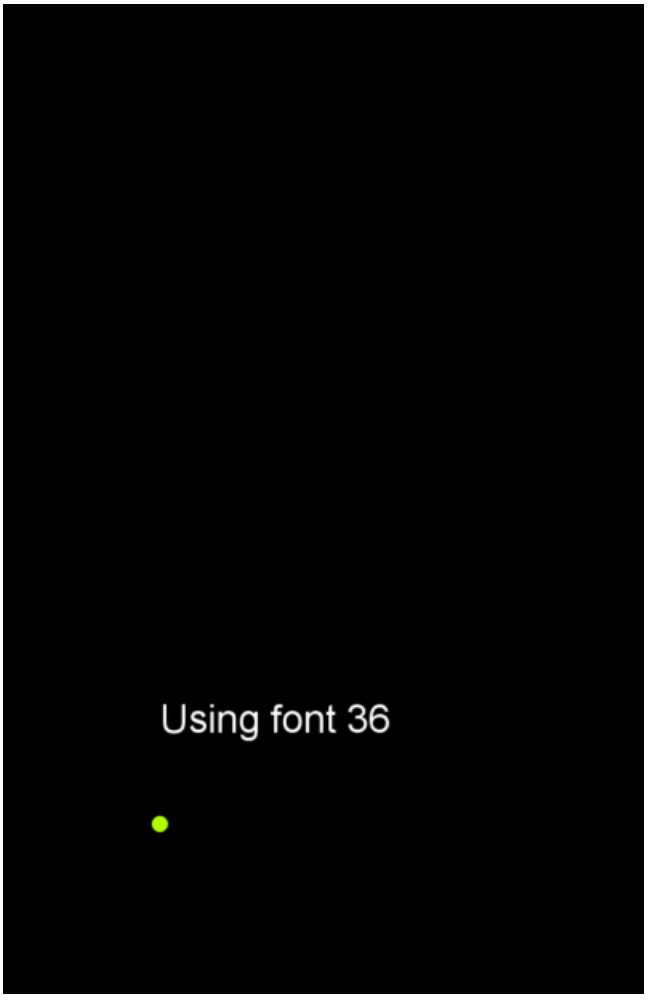
The MonoGame Pipeline Tool provides an easy-to-use environment for converting content files into **.xnb** files for use in CocosSharp and MonoGame applications. For information on content pipelines and why they are useful in game development, see this introduction on content pipelines

This walkthrough will cover the following:

- Installing the MonoGame Pipeline Tool
- Creating a CocosSharp project
- Creating a content project
- Processing files in the MonoGame Pipeline Tool
- Using files at runtime

This walkthrough uses a CocosSharp project to demonstrate how **.xnb** files can be loaded and used in an application. Users of MonoGame will also be able to reference this walkthrough as CocosSharp and MonoGame both use the same **.xnb** content files.

The finished app will display a single sprite displaying a texture from a **.xnb** file and a single label displaying a sprite font from an **.xnb** file:



MonoGame Pipeline Tool discussion

The MonoGame Pipeline Tool is available on Windows, OS X, and Linux. This walkthrough will run the tool on Windows, but it can be followed along on Mac and Linux as well. For information on getting the tool set up on Max or Linux, see this page.

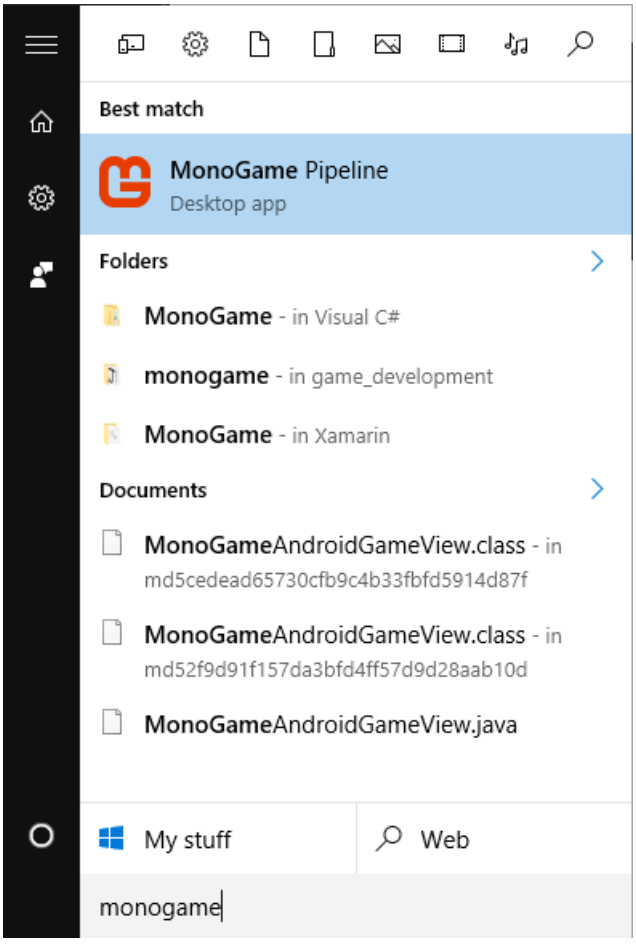
The MonoGame Pipeline Tool is able to create content for iOS applications even when run on Windows, so developers using Xamarin Mac Agent will be able to continue developing on Windows.

Installing the MonoGame Pipeline Tool

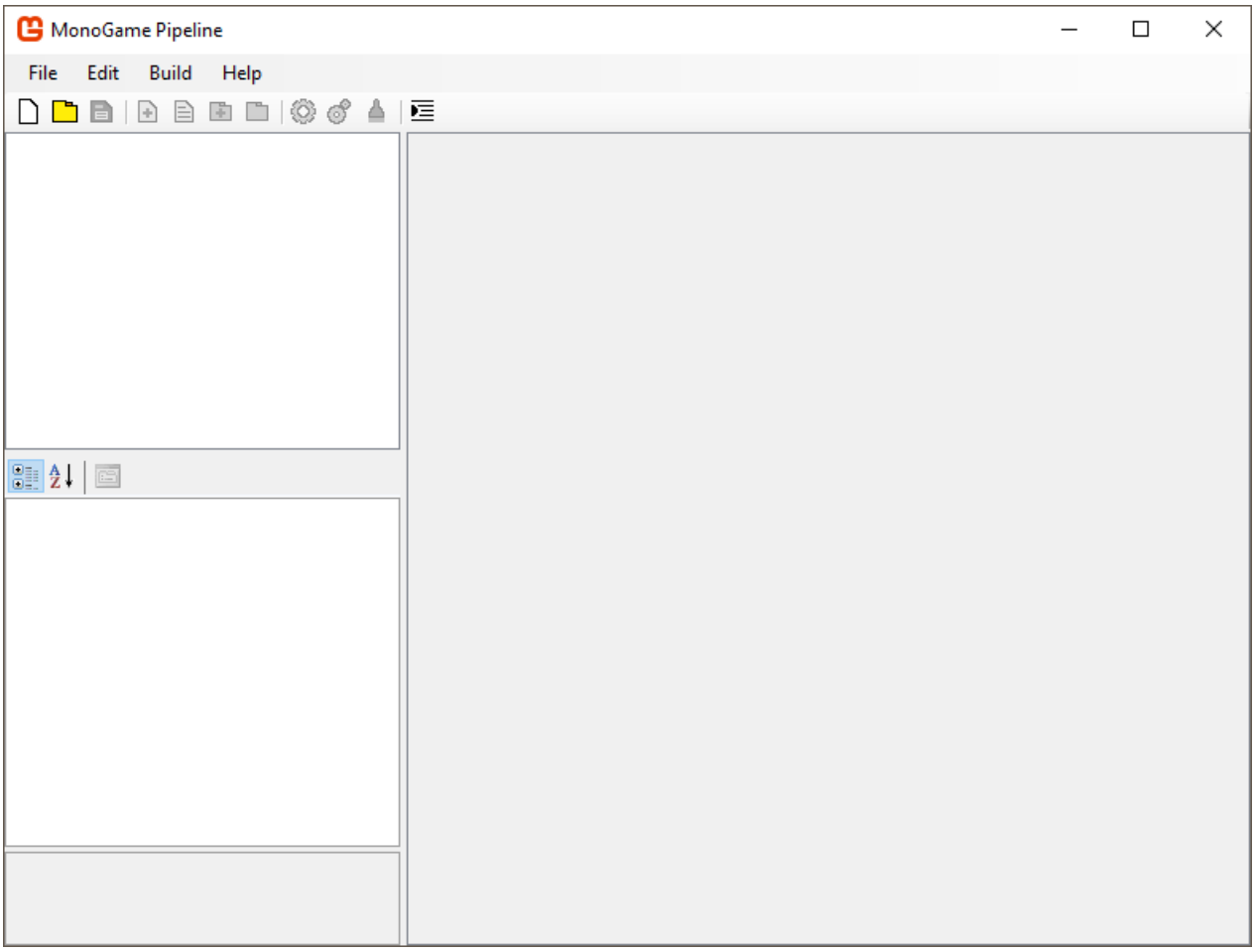
We will begin by installing the MonoGame, which includes the MonoGame Content Pipeline. Note that the MonoGame Content Pipeline is a separate download for Mac. All MonoGame installers can be found on the MonoGame Downloads page. We'll download MonoGame for Visual Studio, but once installed the developer can use MonoGame in Visual Studio for Mac too:

- MonoGame 3.5 for VisualStudio
- MonoGame 3.5 for MacOS
- MonoGame 3.5 Pipeline GUI Tool for MacOS stand alone installer
- MonoGame 3.5 for Linux
- MonoGame 3.5 Source Code On GitHub
- MonoGame 3.5 Assemblies on NuGet

Once downloaded, we'll run through the installer and accept the defaults options. After the installer finishes, the MonoGame Pipeline Tool is installed, and can be found in the Start menu search:



Launch the MonoGame Pipeline Tool:

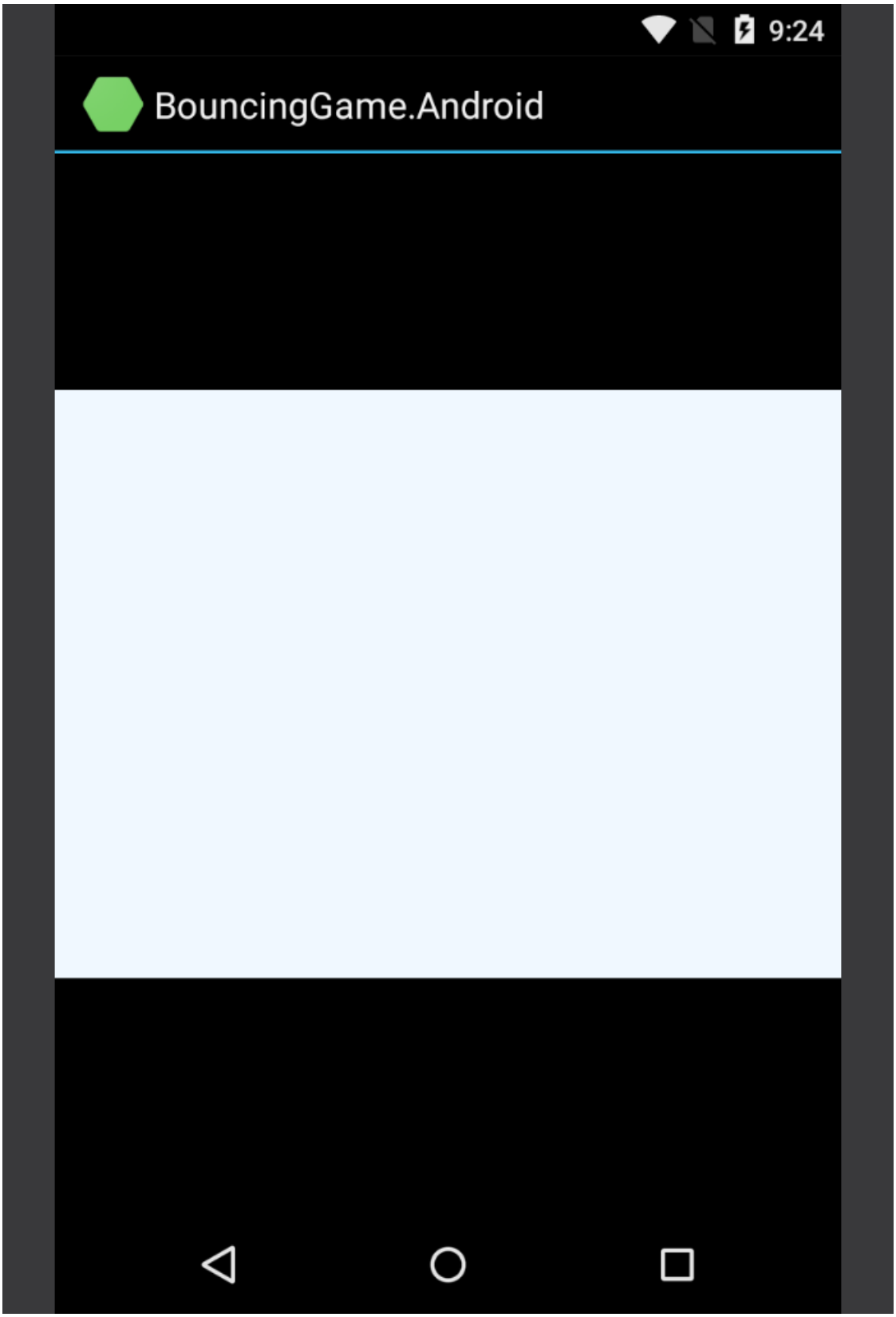


Once the MonoGame Pipeline Tool is running, we can start to make our game and content projects.

Creating an empty CocosSharp project

The next step is to create a CocosSharp project. It's important that we create the CocosSharp project first so that we can save our content project in the folder structure created by the CocosSharp project. To understand the structure of a CocosSharp project, take a look at the BouncingGame, which will be using in this guide. However, if you have an existing CocosSharp project that you'd like to add content to, feel free to use that project instead of BouncingGame.

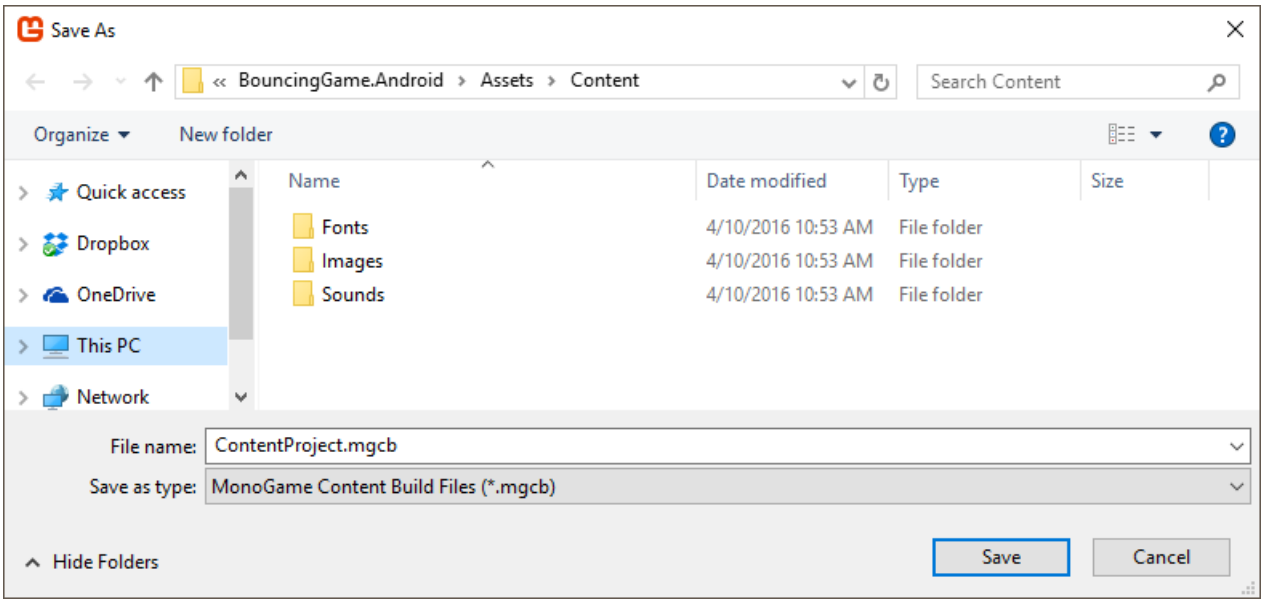
Once the project has been created, we'll run it to verify that it builds and that we have everything set up properly:



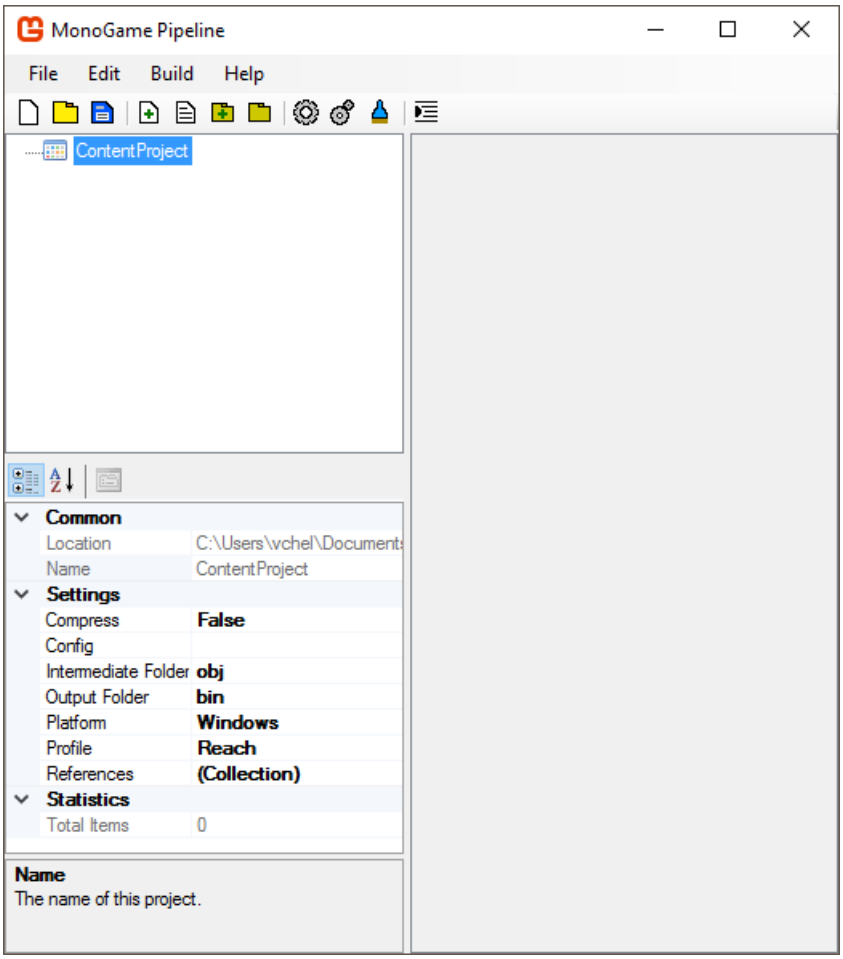
Creating a content project

Now that we have a game project, we can create a MonoGame Pipeline project. To do this, in the MonoGame Pipeline Tool select **File>New...** and navigate to your project's Content folder. For Android, the folder is located at **[project root]\BouncingGame.Android\Assets\Content**. For iOS, the folder is located at **[project root]\BouncingGame.iOS\Content**.

Change the **File name** to **ContentProject** and click the **Save** button:



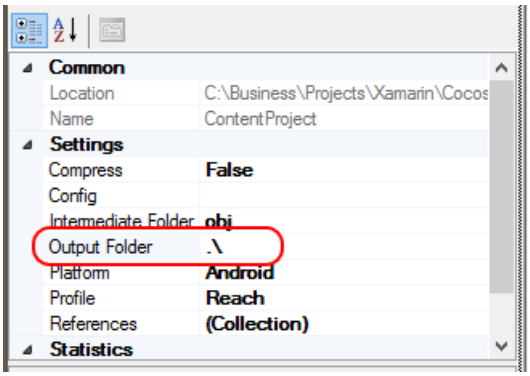
Once the project is created, the MonoGame Pipeline Tool will display information about the project when the root **ContentProject** item is selected:



Let's look at some of the most important options for the content project.

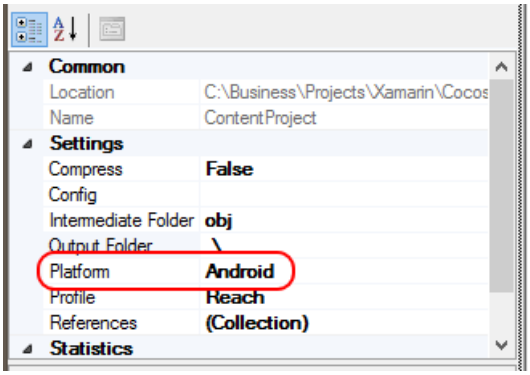
Output folder

This is the folder (relative to the content project itself) where the output **.xnb** files will be saved. To keep things simple, we'll use the same folder to hold our input and output files. In other words we'll change the **Output Folder** to be **.** :



Platform

This defines the target platform for the content. Notice that this is **Windows** by default, so we'll want to change this to our target platform which is **Android** (or iOS if following along with an iOS project).



Processing files in the MonoGame Pipeline Tool

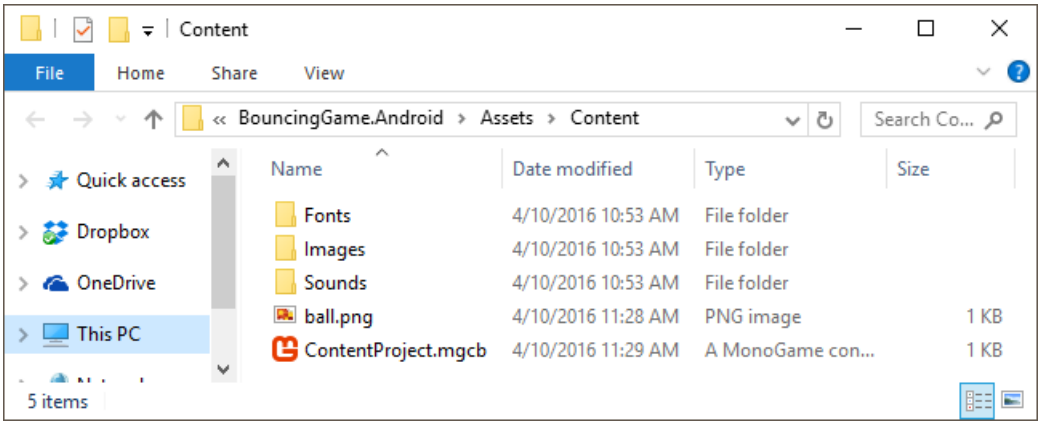
Next, we'll be adding content to our **ContentProject**. For this project, we'll be adding files to the root of the project, but larger projects will typically organize their content in folders.

We'll add two files to our project:

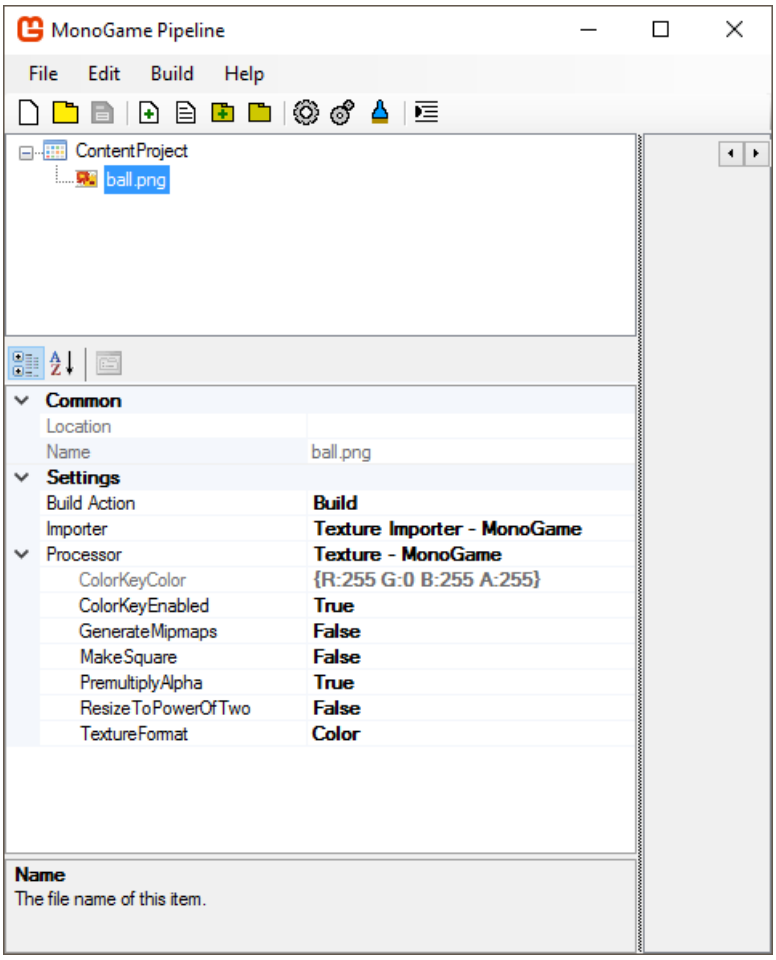
- A **.png** file which will be used to draw a sprite. This file can downloaded here.
- A **.spritefont** file which will be used to draw text on screen. The Content Pipeline Tool supports creating new .spritefont files, so there is no file to download.

Adding a .png file

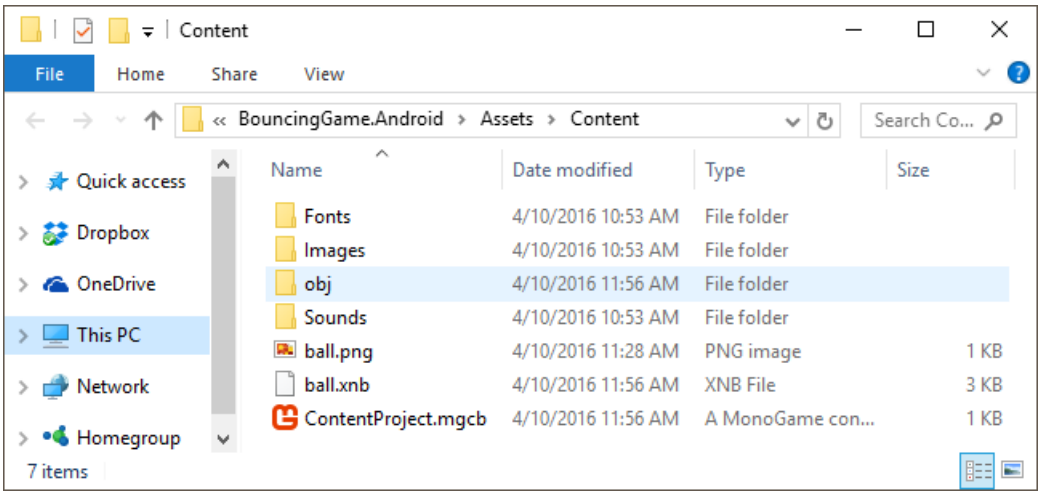
To add a **.png** file to the project, we'll first copy it to the same directory as the pipeline project, which has the **.mgcb** extension.



Next, we'll add the file to the pipeline project. To do this in the MonoGame Pipeline Tool, select **Edit>Add Item...**, select the **ball.png** file and click **Open**. The file will now be part of the content project and, when selected, will display its properties:

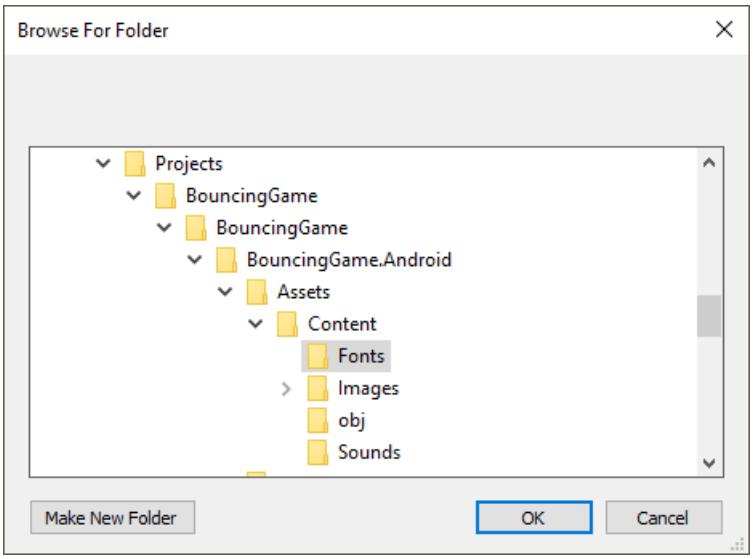


We'll being leaving all the values at their defaults as no changes are required to load the .xnb file in CocosSharp. We can build the file by selecting the **Build>Build** menu option, which will start a build and display output about the build. We can verify that the build worked correctly by checking the **Content** folder for a new **ball.xnb** file:

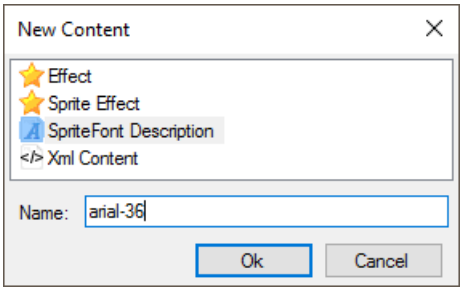


Adding a .spritefont file

We can create .spritefont file through the MonoGame Pipeline Tool. CocosSharp requires fonts to be in a **Fonts** folder, and CocosSharp templates automatically create a Fonts folder automatically. We can add this folder to the MonoGame Pipeline Tool by selecting **Edit>Add>Existing Folder...**. Browse to the **Content** folder and select the **Fonts** folder and click **OK**:



To add a new .sprintefont file, right-click on the Fonts folder and select **Add>New Item...**, select the **SpriteFont Description** option, enter the name **arial-36**, and click **Ok**. CocosSharp requires very specific naming of font files – they must be in the format of [FontType]-[FontSize]. If a font does not match this naming format it will not be loaded by CocosSharp at runtime.



The .spritefont file is actually an XML file which can be edited in any text editor, including Visual Studio for Mac. The most common variables edited in a .spritefont file are the `FontName` and `Size` property:

```
<!-- Modify this string to change the font that will be imported. -->
<FontName>Arial</FontName>

<!-- Size is a float value, measured in points.
Modify this value to change the size of the font. -->
<Size>12</Size>
```

We'll open the file in any text editor. As our **arial-36.spritefont** name suggests, we'll leave the `FontName` as `Arial` but change the `Size` value to `36`:

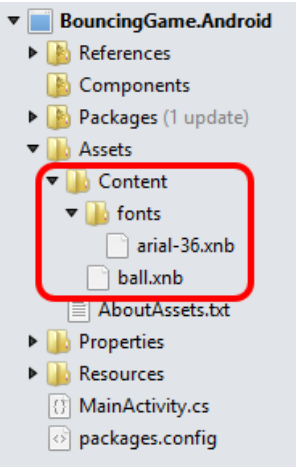
```
<!-- Modify this string to change the font that will be imported. -->
<FontName>Arial</FontName>

<!-- Size is a float value, measured in points.
Modify this value to change the size of the font. -->4/10/2016 12:57:28 PM
<Size>36</Size>
```

Using files at runtime

The .xnb files are now built and ready to be used in our project. We'll be adding the files to Visual Studio for Mac then we'll add code to our `GameScene.cs` file to load these files and display them.

Adding .xnb files to Visual Studio for Mac

First we'll add the files to our project. In Visual Studio for Mac, we'll expand the **BouncingGame.Android** project, expand the **Assets** folder, right-click on the **Content** folder, and select **Add>Add Files....** First, we'll select the **ball.xnb** we built earlier and click **Open**. Then repeat the above steps, but add the **arial-36.xnb** file. We'll select the **Keep the file in its current subdirectory** option if Visual Studio for Mac asks how to add the file. Once finished both files should be part of our project:



Adding GameScene.cs

We'll create a class called `GameScene,` which will contain our sprite and text objects. To do this, right-click on the **BouncingGame** (not BouncingGame.Android) project and select **Add>New File...**. Select the **General** category, select the **Empty Class** option, and then enter the name **GameScene**.

Once created, we'll modify the `GameScene.cs` file to contain the following code:

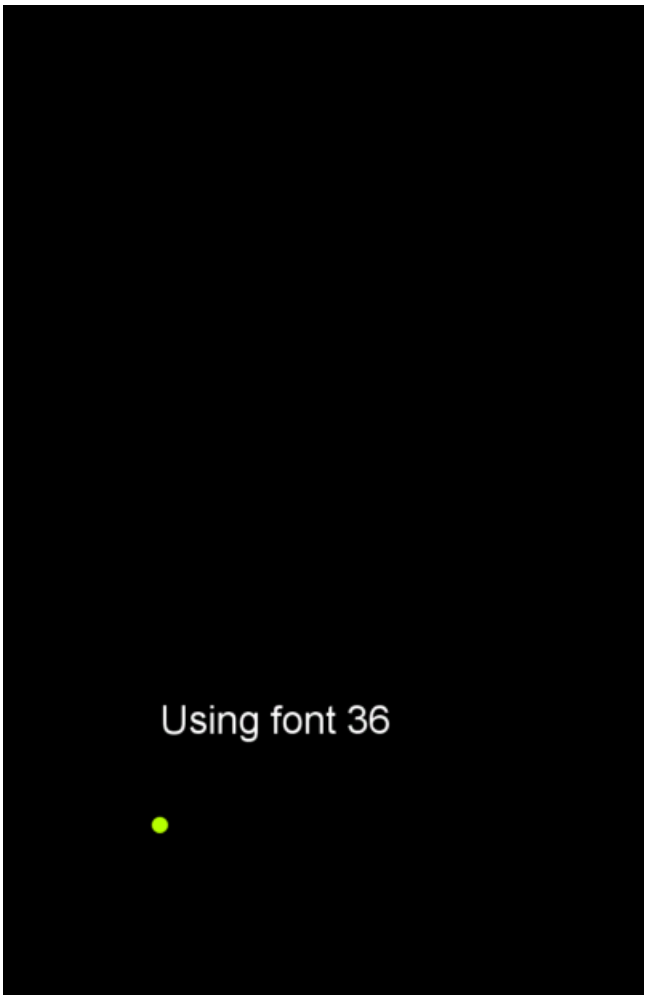
```
using System;
using CocosSharp;

namespace BouncingGame
{
    public class GameScene : CCScene
    {
        // All visual elements must be added to a CCLayer:
        CCLayer mainLayer;

        // The CCSprite is used to display the "ball" texture
        CCSprite sprite;
        // The CCLabelTtf is used to display the Arial36 sprite font
        CCLabelTtf label;

        public GameScene(CCWindow mainWindow) : base(mainWindow)
        {
            // Instantiate the CCLayer first:
            mainLayer = new CCLayer ();
            AddChild (mainLayer);

            // Now we can create the Sprite using the ball.xnb file:
            sprite = new CCSprite ("ball");
            sprite.PositionX = 200;
            sprite.PositionY = 200;
            mainLayer.AddChild (sprite);

            // The font name (arial) and size (36) need to match
            // the .spritefont definition and file name.
            label = new CCLabelTtf ("Using font 36", "arial", 36);
            label.PositionX = 200;
            label.PositionY = 300;
            mainLayer.AddChild (label);
        }
    }
}
```

We won't be discussing the code above since working with CocosSharp visual objects like CCSprite and CCLabelTtf is covered in the BouncingGame guide.

We also need to add code to load our newly-created `GameScene`. To do this we'll open the `GameAppDelegate.cs` file (which is located in the **BouncingGame** PCL) and modify the `ApplicationDidFinishLaunching` method so it looks like:

```
public override void ApplicationDidFinishLaunching (CCApplication application, CCWindow mainWindow)
{
    application.PreferMultiSampling = false;
    application.ContentRootDirectory = "Content";

    // New code:
    GameScene gameScene = new GameScene (mainWindow);
    mainWindow.RunWithScene (gameScene);
}
```

When running, our game will look like:



Summary

This walkthrough showed how to use the MonoGame Pipeline Tool to create .xnb files from an input .png file as well as how to create a new .xnb file from a newly-created .sprintefont file. It also discussed how to structure CocosSharp projects to use .xnb files and how to load these files at runtime.

Related links

- MonoGame Downloads
- MonoGame Pipeline Documentation
- Starting BouncingGame Project for Android (sample)
- ball.png Graphic (sample)

Improving frame rate with CCSpriteSheet

10/3/2018 • 6 minutes to read • Edit Online

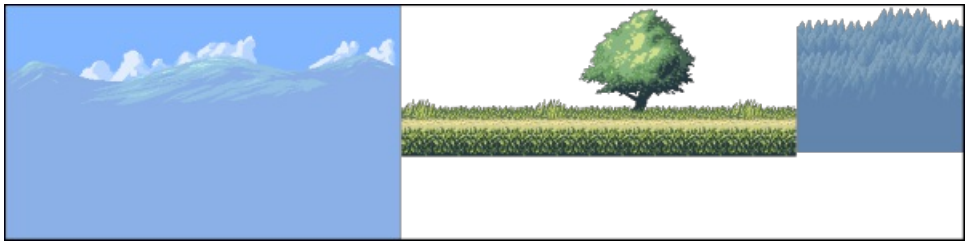
CCSpriteSheet provides functionality for combining and using many image files in one texture. Reducing texture count can improve a game's load times and framerate.

Many games require optimization efforts to run smoothly and load quickly on mobile hardware. The `CCSpriteSheet` class can help address many common performance problems encountered by CocosSharp games. This guide covers common performance problems and how to address them using the `CCSpriteSheet` class.

What is a sprite sheet?

A *sprite sheet*, which can also be referred to as a *texture atlas*, is an image which combines multiple images into one file. This can improve runtime performance as well as content load times.

For example, the following image is a simple sprite sheet created by three separate images. The individual images can be any size, and the resulting sprite sheet is not required to be completely filled:

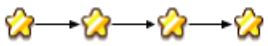


Render states

Visual CocosSharp objects (such as `CCSprite`) simplify rendering code over traditional graphical API rendering code such as MonoGame or OpenGL, which require the creation of vertex buffers (as outlined in the Drawing 3D Graphics with Vertices in MonoGame guide). Despite its simplicity, CocosSharp does not eliminate the cost of setting *render states*, which are the number of times that the rendering code must switch textures or other rendering-related states.

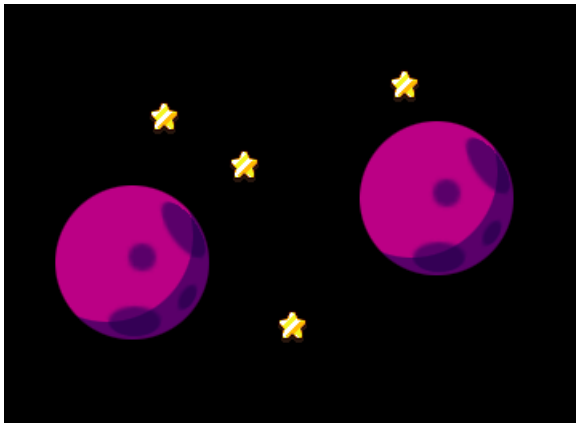
CocosSharp's internal code renders each visual element sequentially, by traversing the visual tree beginning with the current `CCScene`. For example, consider the following scene:

CocosSharp would render the four stars in sequence:

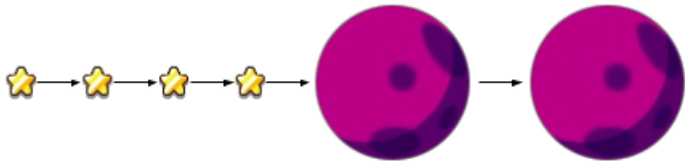


Since each `CCSprite` uses the same texture, CocosSharp can group all four stars together. This code requires only one render state assignment (assignment of the star texture) per frame. This scenario is very efficient.

Of course, very few games use only one image. The following scene introduces a planet graphic:

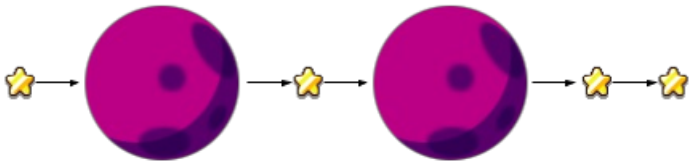


Ideally CocosSharp should draw all sprites using one image first (such as the stars), then the remainder of the sprites using the other image (the planet):



The ordering above requires two render states: one on the first star, one on the first planet.

If all `CCSprite` instances are children of the same `CCNode`, then CocosSharp will optimize the draw order to reduce render state changes. If, on the other hand, the `CCSprite` instances are organized such that CocosSharp is unable to optimize the rendering (such as if they are part of different entity `CCNode` instances), then the order may not be optimal. The following shows the worst possible draw order, resulting in five render states:



Render states can be difficult to optimize because the draw order must obey the visual tree of `CCNode` instances. This tree is often structured to be easy to work with (such as entities containing their visual children), or organized due to the desired visual layout as defined by an artist.

Of course, the ideal situation is to have a single render state, despite having multiple images. CocosSharp games can accomplish this by combining all images into a single file, then loading that one file (along with its accompanying **.plist** file) into a `CCSpriteSheet`. Using the `CCSpriteSheet` class becomes even more important for games which have a large number of images, or which have very complicated layouts.

Load times

Combining multiple images into one file also improves a game's load times for a number of reasons:

- Combining multiple images into a single file can reduce the overall number of pixels used through efficient packing
- Loading fewer files means less per-file overhead, such as parsing .png headers
- Loading fewer files requires less seek time, which is important for disk-based media such as DVDs and traditional computer hard drives

Using CCSpriteSheet in code

To create a `CCSpriteSheet` instance, the code must supply an image and a file which defines the regions of the

image to use for each frame. The image can be loaded as a **.png** or **.xnb** file (if using the Content Pipeline). The file defining the frames is a **.plist** file which can be created by hand or *TexturePacker* (which we'll discuss below).

The sample app, which can be downloaded here, creates the `CCSpriteSheet` from a **.png** and **.plist** file using the following code:

```
CCSpriteSheet sheet = new CCSpriteSheet ("sheet.plist", "sheet.png");
```

Once loaded, the `CCSpriteSheet` contains a `List` of `CCSpriteFrame` instances – each instance corresponding to one of the source images used to create the entire sheet. In the case of the **SpriteSheetDemo** project, the `CCSpriteSheet` contains three images. The **.plist** file can be inspected in Visual Studio for Mac or in any text editor to see which images are available. If we view the **.plist** file in a text editor we can see the three frames (sections omitted to emphasize the key names):

```
...
<dict>
 <key>frames</key>
 <dict>
  <key>farBackground.png</key>
  ...
  <key>foreground.png</key>
  ...
<key>forestBackground.png</key>
...
```

We can use the `Find` method to find frames by name, as follows (code omitted to emphasize `CCSpriteFrame` usage):

```
CCSpriteFrame frame;
...
frame = sheet.Frames.Find(item=>item.TextureFilename == "farBackground.png");
CCSprite sprite = new CCSprite (frame);
...
```

Since the `CCSprite` constructor can take a `CCSpriteFrame` parameter, the code never has to investigate the details of the `CCSpriteFrame`, such as which texture it uses, or the region of the image in the master sprite sheet.

Creating a sprite sheet .plist

The .plist file is an xml-based file, which can be created and edited by hand. Similarly, image editing programs can be used to combine multiple files into one larger file. Since creating and maintaining sprite sheets can be very time consuming, we will look at the TexturePacker program which can export files in the CocosSharp format. TexturePacker offers a free and a "Pro" version, and is available for Windows and Mac OS. The remainder of this guide can be followed by using the free version.

TexturePacker can be downloaded from the TexturePacker website. When opened, TexturePacker does not have a project loaded. The starting screen allows you to add sprites, open recent projects (if other projects have been created), and select the format to use for the sprite sheet. CocosSharp uses the Cocos2D Data Format:

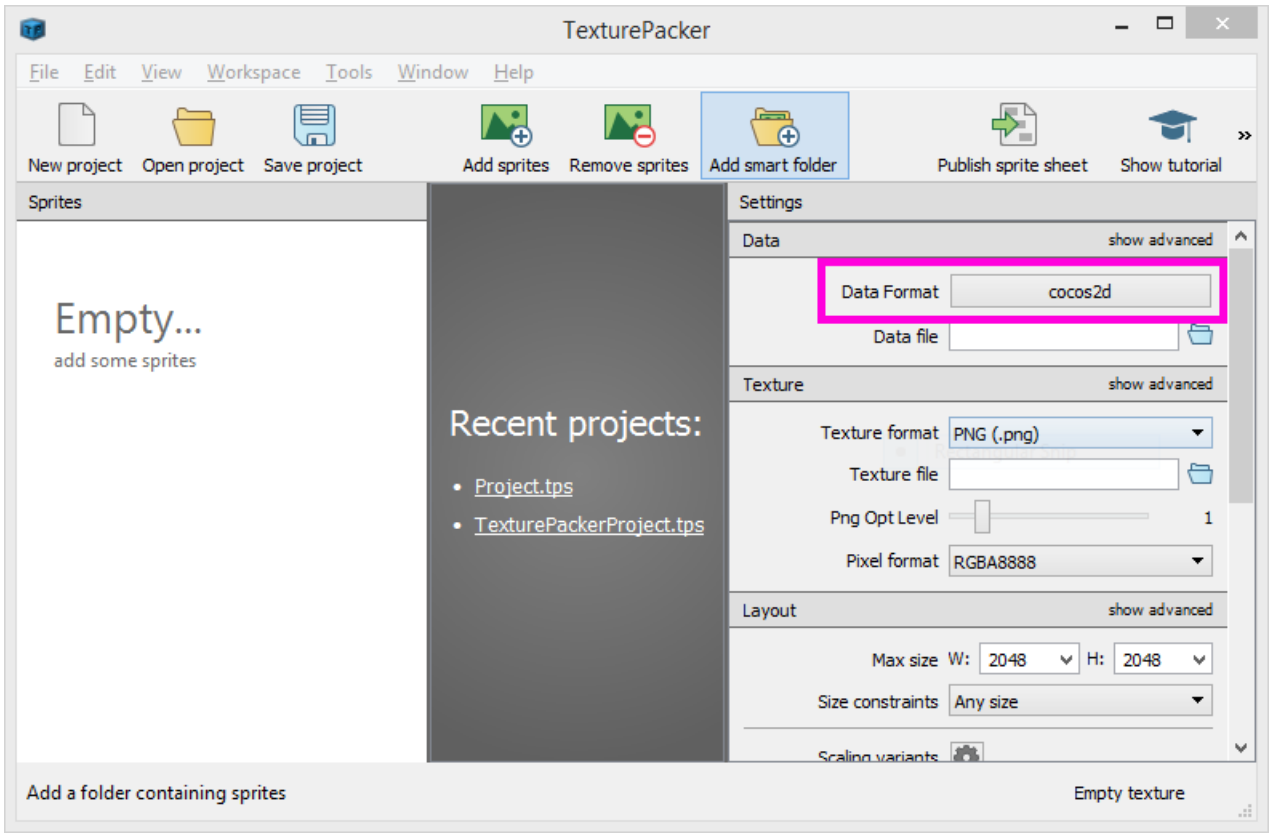
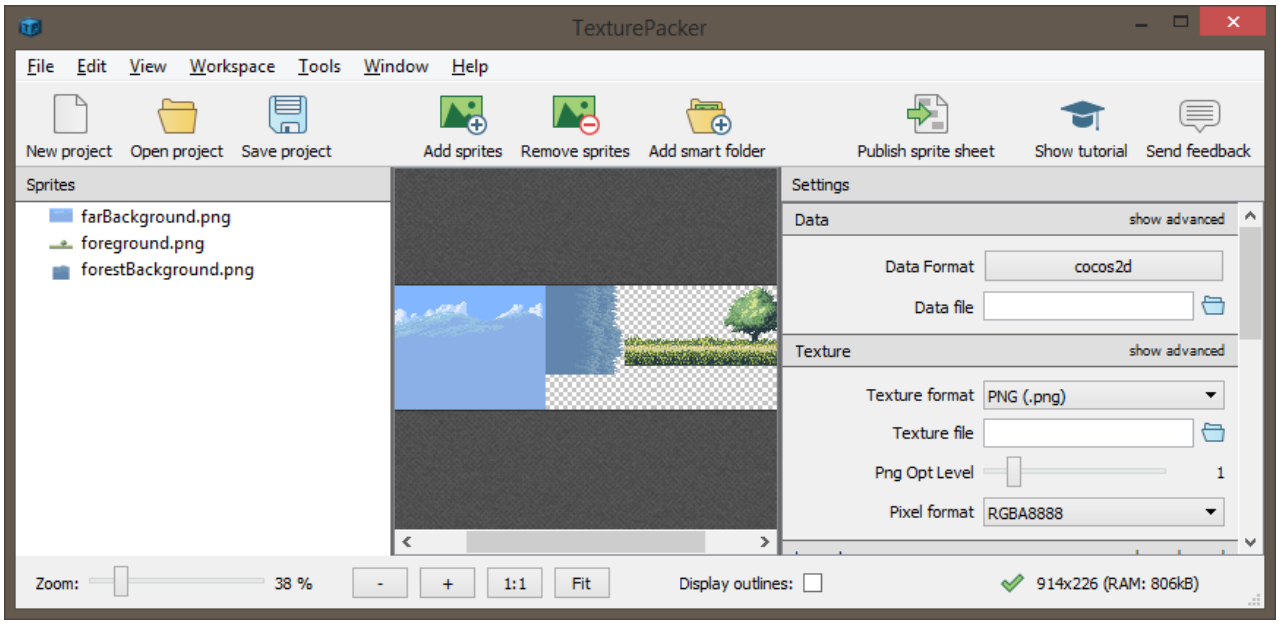


Image files (such as **.png**) can be added to TexturePacker by drag-dropping them from Windows Explorer on Windows or Finder on Mac. TexturePacker automatically updates the sprite sheet preview whenever a file is added:



To export a sprite sheet, click the **Publish sprite sheet** button and select a location for the sprite sheet. TexturePacker will save a .plist file and an image file.

To use the resulting files, add both the .png and .plist to a CocosSharp project. For information on adding files to CocosSharp projects, see the BouncingGame guide. Once the files are added, they can be loaded into a `CCSpriteSheet` as was shown earlier in the code above:

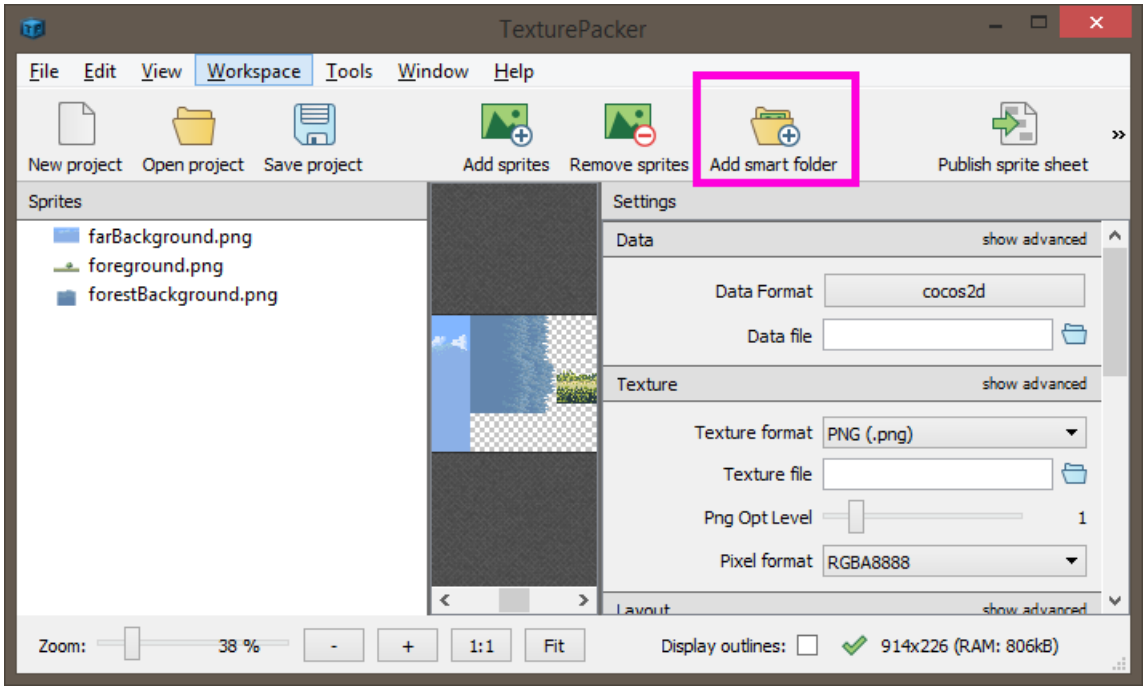
```
CCSpriteSheet sheet = new CCSpriteSheet ("sheet.plist", "sheet.png");
```

Considerations for maintaining a TexturePacker sprite sheet

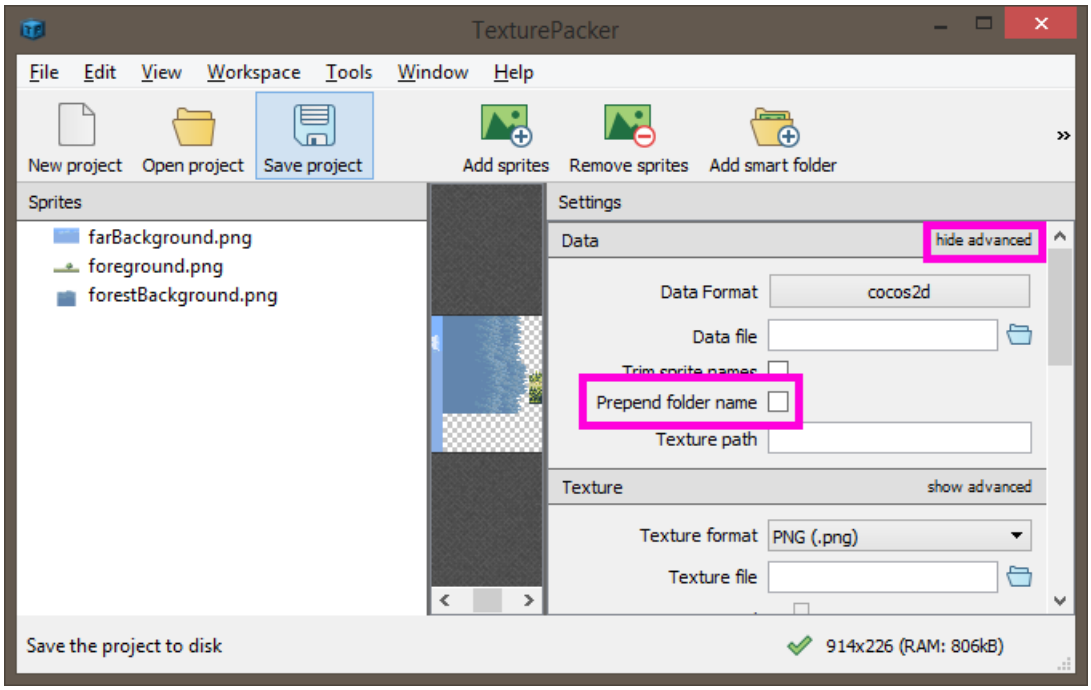
As games are developed, artists may add, remove, or modify art. Any change requires an updated sprite sheet. The

following considerations can ease sprite sheet maintenance:

- Keep the original files (the files used to create the sprite sheets) in a folder in your project, and make sure they are added to version control. These files will be needed to re-create the sprite sheet whenever a change is made.
- Do not add the original files to Visual Studio for Mac/Visual Studio, or if they are added, set the **Build Action** to **None**. If the files are added and have the platform-specific **Build Action**, then they will needlessly increase the resulting app's file size.
- Consider using *smart folders* in TexturePacker. Smart folders automatically add any contained images to the sprite sheet. This feature can save a lot of time when developing games with a large number of images.



- Keep an eye on sprite sheet texture sizes. Some older phone hardware does not support texture sizes larger than 2048x2048. Also, a 32-bit image of 2048x2048 uses nearly 17 mbytes of RAM – a significant amount of memory.
- TexturePacker does not include folders in sprite names by default, so name conflicts are possible. It's best to decide whether to include folder names or not at the beginning of development. Larger games should consider using folder names to prevent conflicts. To include folder paths, click **show advanced** in the **Data** section and check **Prepend folder name**.



Summary

This guide covers how to create and use the `CCSpriteSheet` class. It also covers how to construct files which can be loaded into `CCSpriteSheet` instances using the TexturePacker program.

Related links

- CCSpriteSheet
- Full Demo (sample)

Texture caching using CCTextureCache

10/3/2018 • 6 minutes to read • Edit Online

CocosSharp's CCTextureCache class provides a standard way to organize, cache, and unload content. It is especially useful for large games which may not fit entirely into RAM, simplifying the process of grouping and disposing of textures.

The `CCTextureCache` class is an essential part of CocosSharp game development. Most CocosSharp games use the `CCTextureCache` object, even if not explicitly, as many CocosSharp methods internally use a *shared* texture cache.

This guide covers the `CCTextureCache` and why it is important for game development. Specifically it covers:

- Why texture caching matters
- Texture lifespan
- Using SharedTextureCache
- Lazy loading vs. pre-loading with AddImage
- Disposing textures

Why texture caching matters

Texture caching is an important consideration in game development as texture loading is a time-consuming operation and textures require a significant amount of RAM at runtime.

As with any file operation, loading textures from disk can be a costly operation. Texture loading can take extra time if the file being loaded requires processing, such as being decompressed (as is the case for png and jpg images). Texture caching can reduce the number of times that the application must load files from disk.

As mentioned above, textures also occupy a large amount of runtime memory. For example a background image sized to the resolution of an iPhone 6 (1344x750) would occupy 4 megabytes of RAM – even if the PNG file is only few kilobytes in size. Texture caching provides a way to share texture references within an app and also an easy way to unload all content when transitioning between different game states.

Texture lifespan

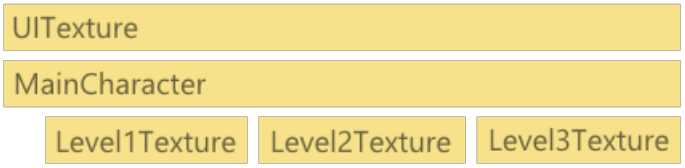
CocosSharp textures may be kept in memory for the entire length of an app's execution, or they may be short lived. To minimize memory usage an app should dispose of textures when no longer needed. Of course, this means that textures may be disposed and re-loaded at a later time, which can increase load times or hurt performance during loads.

Texture loading often requires a tradeoff between memory usage and load times/runtime performance. Games which use a small amount of texture memory can keep all textures in memory as needed, but larger games may need to unload textures to free up space.

The following diagram shows a simple game which loads textures as needed and keeps them in memory for the entire length of execution:

The first two bars represent textures which are needed immediately upon the game's execution. The following three bars represent textures for each level, loaded as needed.

If the game was large enough it would eventually load enough textures to fill all RAM provided by the device and OS. To solve this, a game may unload texture data when it is no longer needed. For example, the following diagram shows a game which unloads Level1Texture when it is no longer needed, then loads Level2Texture for the next level. The end result is that only three textures are held in memory at any given time:



The diagram shown above indicates that texture memory usage can be reduced by unloading, but this may require additional loading times if a player decides to replay a level. It's also worth noting that the UITexture and MainCharacter textures are loaded and never unloaded. This implies that these textures are needed in all levels, so they are always kept in memory.

Using SharedTextureCache

CocosSharp automatically caches textures when loading them through the `CCSprite` constructor. For example the following code only creates one texture instance:

```
for (int i = 0; i < 100; i++)
{
    CCSprite starSprite = new CCSprite ("star.png");
    starSprite.PositionX = i * 32;
    this.AddChild (starSprite);
}
```

CocosSharp automatically caches the `star.png` texture to avoid the expensive alternative of creating numerous identical `CCTexture2D` instances. This is accomplished by `AddImage` being called on a shared `CCTextureCache` instance, specifically `CCTextureCache.SharedTextureCache.Shared`. To understand how the `SharedTextureCache` is used we can look at the following code which is functionally identical to calling the `CCSprite` constructor with a string parameter:

```
CCSprite starSprite = new CCSprite ();
 starSprite.Texture = CCTextureCache.SharedTextureCache.AddImage ("star.png");
```

`AddImage` checks if the argument file (in this case `star.png`) has already been loaded. If so, then the cached instance is returned. If not then it is loaded from the file system, and a reference to the texture is stored internally for subsequent `AddImage` calls. In other words the `star.png` image is only loaded once, and subsequent calls require no additional disk access or additional texture memory.

Lazy loading vs. pre-loading with AddImage

`AddImage` allows code to be written the same whether the requested texture is already loaded or not. This means that content will not be loaded until it is needed; however, this can also cause performance problems at runtime due to unpredictable content loading.

For example consider a game where the player's weapon can be upgraded. When upgraded, the weapon and projectiles will visibly change, resulting in new textures being used. If the content is lazy-loaded then the textures associated with upgraded weapons will not be loaded initially, but rather at a later time when the player acquires the upgrades.

This mid-gameplay loading can cause the game to *pop*, which is a short but noticeable freeze in execution. To prevent this, the code can predict which textures may be needed up front and pre-load them. For example, the following may be used to pre-load textures:

```
void PreLoadImages()
{
    var cache = CCTextureCache.SharedTextureCache;

    cache.AddImage ("powerup1.png");
    cache.AddImage ("powerup2.png");
    cache.AddImage ("powerup3.png");

    cache.AddImage ("enemy1.png");
    cache.AddImage ("enemy2.png");
    cache.AddImage ("enemy3.png");

    // pre-load any additional content here to
    // prevent pops at runtime
}
```

This pre-loading can result in wasted memory and can increase startup time. For example, the player may never actually obtain a power-up represented by the `powerup3.png` texture, so it will be unnecessarily loaded. Of course this may be a necessary cost to pay to avoid a potential pop in gameplay, so it's usually best to preload content if it will fit in RAM.

Disposing textures

If a game does not require more texture memory than is available on the minimum spec device then textures do not need to be disposed. On the other hand, larger games may need to free up texture memory to make room for new content. For example a game may use a large amount of memory storing textures for an environment. If the environment is only used in a specific level then it should be unloaded when the level ends.

Disposing a single texture

Removing a single texture first requires calling the `Dispose` method, then manual removal from the `CCTextureCache`.

The following shows how to completely remove a background sprite along with its texture:

```
void DisposeBackground()
{
    // Assuming this is called from a CCLayer:
    this.RemoveChild (backgroundSprite);

    CCTextureCache.SharedTextureCache.RemoveTexture (backgroundsprite.Texture);

    backgroundSprite.Texture.Dispose ();
}
```

Directly disposing textures can be effective when dealing with a small number of textures but this can become error-prone when dealing with larger texture sets.

Textures can be grouped into custom (non-shared) `CCTextureCache` instances to simplify texture cleanup.

For example, consider an example where content is preloaded using a level-specific `CCTextureCache` instance. The `CCTextureCache` instance may be defined in the class defining the level (which may be a `CCLayer` or `CCScene`):

```
CCTextureCache levelTextures;
```

The `levelTextures` instance can then be used to preload the level-specific textures:

```
void PreloadLevelTextures(CCApplication application)
{
    levelTextures = new CCTextureCache (application);

    levelTextures.AddImage ("Background.png");
    levelTextures.AddImage ("Foreground.png");
    levelTextures.AddImage ("Enemy1.png");
    levelTextures.AddImage ("Enemy2.png");
    levelTextures.AddImage ("Enemy3.png");

    levelTextures.AddImage ("Powerups.png");
    levelTextures.AddImage ("Particles.png");
}
```

Finally when the level ends, the textures can be all disposed at once through the `CCTextureCache`:

```
void EndLevel()
{
    levelTextures.Dispose ();
    // Perform any other end-level cleanup
}
```

The Dispose method will dispose all internal textures, clearing out the memory used by these textures. Combining `CCTextureCache.Shared` with a level or game mode-specific `CCTextureCache` instance results in some textures persisting through the entire game, and some being unloaded as levels end, similar to the diagram presented at the beginning of this guide:

Summary

This guide shows how to use the `CCTextureCache` class to balance memory usage and runtime performance. `CCTexturCache.SharedTextureCache` can be explicitly or implicitly used to load and cache textures for the life of the application, while `CCTextureCache` instances can be used to unload textures to reduce memory usage.

Related links

- https://github.com/mono/CocosSharp
- /api/type/CocosSharp.CCTextureCache/

2D math with CocosSharp

10/3/2018 • 12 minutes to read • Edit Online

This guide covers 2D mathematics for game development. It uses CocosSharp to show how to perform common game development tasks and explains the math behind these tasks.

To position and move objects with code is a core part of developing games of all sizes. Positioning and moving require the use of math, whether a game requires moving an object along a straight line, or the use of trigonometry for rotation. This document will cover the following topics:

- Velocity
- Acceleration
- Rotating CocosSharp objects
- Using rotation with velocity

Developers who do not have a strong math background, or who have long-forgotten these topics from school, do not need to worry – this document will break concepts down into bite-sized pieces, and will accompany theoretical explanations with practical examples. In short, this article will answer the age-old math student question: "When will I actually need to use this stuff?"

Requirements

Although this document focuses primarily on the mathematical side of CocosSharp, code samples assume working with objects inheriting form `CCNode`. Furthermore, since `CCNode` does not include values for velocity and acceleration, the code assumes working with Entities which provide values such as VelocityX, VelocityY, AccelerationX, and AccelerationY. For more information on entities, see our walkthrough on Entities in CocosSharp.

Velocity

Game developers use the term *velocity* to describe how an object is moving – specifically how fast something is moving and the direction that it is moving.

Velocity is defined using two types of units: a position unit and a time unit. For example, a car's speed is defined as miles per hour or kilometers per hour. Game developers often use pixels per second to define how fast an object moves. For example, a bullet may move at a speed of 300 pixels per second. That is, if a bullet is moving at 300 pixels per second, then it will have moved 600 units in two seconds, and 900 units in three seconds, and so on. More generally, the velocity value *adds* to the position of an object (as we'll see below).

Although we used speed to explain the units of velocity, the term speed typically refers to a value independent of direction, while the term velocity refers to both speed and direction. Therefore, the assignment of a bullet's velocity (assuming bullet is a class which includes the necessary properties) may look like this:

```
// This bullet is not moving horizontally, so set VelocityX to 0:
bulletInstance.VelocityX = 0;
// Positive Y is "up" so move the bullet up 300 units per second:
bulletInstance.VelocityY = 300;
```

Implementing velocity

CocosSharp does not implement velocity, so objects requiring movement will need to implement their own movement logic. New game developers implementing velocity often make the mistake of making their velocity

dependent on frame rate. That is, the following *incorrect implementation* will seem to provide correct results, but will be based on the game's frame rate:

```
// VelocityX and VelocityY will be added every time this code executes
this.PositionX += this.VelocityX;
this.PositionY += this.VelocityY;
```

If the game runs at a higher frame rate (such as 60 frames per second instead of 30 frames per second), then the object will appear to move faster than if running at a slower frame rate. Similarly, if the game is unable to process frames at as high of a frame rate (which may be caused by background processes using the device's resources), the game will appear to slow down.

To account for this, velocity is often implemented using a time value. For example, if the `seconds` variable represents the number (or fraction) of seconds since the last time velocity was applied, then the following code would result in the object having consistent movement regardless of frame rate:

```
// VelocityX and VelocityY will be added every time this code executes
this.PositionX += this.VelocityX * seconds;
this.PositionY += this.VelocityY * seconds;
```

Consider that a game which runs at a lower frame rate will update the position of its objects less frequently. Therefore, each update will result in the objects moving further than they would if the game were updating more frequently. The `seconds` value accounts for this, by reporting how much time has passed since the last update.

For an example of how to add time-based movement, see this recipe covering time based movement.

Calculating positions using velocity

Velocity can be used to make predictions about where an object will be after some amount of time passes, or to help tune objects' behavior without needing to run the game. For example, a developer who is implementing the movement of a fired bullet needs to set the bullet's velocity after it is instantiated. The screen size can be used to provide a basis for setting velocity. That is, if the developer knows that the bullet should move the height of the screen in 2 seconds, then the velocity should be set to the height of the screen divided by 2. If the screen is 800 pixels tall, then the bullet's speed would be set to 400 (which is 800/2).

Similarly, in-game logic may need to calculate how long an object will take to reach a destination given its velocity. This can be calculated by dividing the distance to travel by the travelling object's velocity. For example, the following code shows how to assign text to a label which displays how long until a missile reaches its target:

```
// We'll assume only the X axis for this example
float distanceX = target.PositionX - missile.PositionX;

float secondsToReachTarget = distanceX / missile.VelocityX;

label.Text = secondsToReachTarget + " seconds to reach target";
```

Acceleration

Acceleration is a common concept in game development, and it shares many similarities with velocity. Acceleration quantifies whether an object is speeding up or slowing down (how the velocity value changes over time). Acceleration *adds* to velocity, just like velocity adds to position. Common applications of acceleration include gravity, a car speeding up, and a space ship firing its thrusters.

Similar to velocity, acceleration is defined in a position and time unit; however, acceleration's time unit is referred to as a *squared* unit, which reflects how acceleration is defined mathematically. That is, game acceleration is often

measured in *pixels per second squared*.

If an object has an X acceleration of 10 units per second squared, then that means that its velocity will increase by 10 every second. If starting from a standstill, after one second it will be moving at 10 units per second, after two seconds 20 units per second, and so on.

Acceleration in two dimensions requires an X and Y component, so it may be assigned as follows:

```
// No horizontal acceleration:
icicle.AccelerationX = 0;
// Simulate gravity with Y acceleration. Negative Y is down, so assign a negative value:
icicle.AccelerationY = -50;
```

Acceleration vs. deceleration

Although acceleration and deceleration are sometimes differentiated in every-day speech, there is no technical difference between the two. Gravity is a force which results in acceleration. If an object is thrown upward then gravity will slow it down (decelerating), but once the object has stopped climbing and is falling in the same direction as gravity then gravity is speeding it up (accelerating). As shown below, the application of an acceleration is the same whether it is being applied in the same direction or opposite direction of movement.

Implementing acceleration

Acceleration is similar to velocity when implementing – it is not automatically implemented by CocosSharp, and time-based acceleration is the desired implementation (as opposed to frame-based acceleration). Therefore a simple acceleration (along with velocity) implementation may look like:

```
this.VelocityX += this.AccelerationX * seconds;
this.VelocityY += this.AccelerationY * seconds;
this.PositionX += this.VelocityX * seconds;
this.PositionY += this.VelocityY * seconds;
```

The code above is what is referred to as a *linear approximation* for acceleration implementation. Effectively, it implements acceleration with a fairly close degree of accuracy, but it is not a perfectly accurate model of acceleration. It is included above to help explain the concept of how acceleration is implemented.

The following implementation is a mathematically accurate application of acceleration and velocity:

```
float halfSecondsSquared = (seconds * seconds) / 2.0f;

this.PositionX +=
    this.Velocity.X * seconds + this.AccelerationX * halfSecondsSquared;
this.PositionY +=
    this.Velocity.Y * seconds + this.AccelerationY * halfSecondsSquared;

this.VelocityX += this.AccelerationX * seconds;
this.VelocityY += this.AccelerationY * seconds;
```

The most obvious difference to the code above is the `halfSecondsSquared` variable and its usage to apply acceleration to position. The mathematical reason for this is beyond the scope of this tutorial, but developers interested in the math behind this can find more information in this discussion about integrating acceleration.

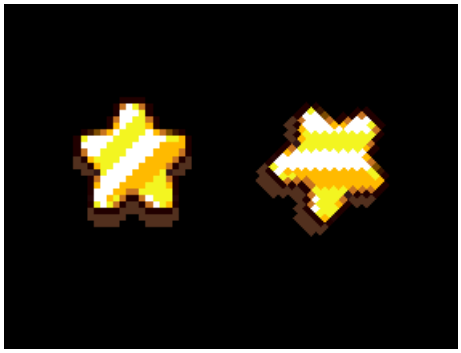
The practical impact of `halfSecondSquare` is that acceleration will behave mathematically accurately and predictably regardless of frame rate. The linear approximation of acceleration is subject to frame rate – the lower the framerate drops the less accurate the approximation becomes. Using `halfSecondsSquared` guarantees that code will behave the same regardless of framerate.

Angles and rotation

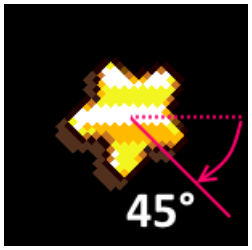
Visual objects such as `CCSprite` support rotation through a `Rotation` variable. This can be assigned to a value to set its rotation in degrees. For example, the following code shows how to rotate a `CCSprite` instance:

```
CCSprite unrotatedSprite = new CCSprite("star.png");
unrotatedSprite.IsAntialiased = false;
unrotatedSprite.PositionX = 100;
unrotatedSprite.PositionY = 100;
this.AddChild (unrotatedSprite);

CCSprite rotatedSprite = new CCSprite("star.png");
rotatedSprite.IsAntialiased = false;
// This sprite is moved to the right so it doesn't overlap the first
rotatedSprite.PositionX = 130;
rotatedSprite.PositionY = 100;
rotatedSprite.Rotation = 45;
this.AddChild (rotatedSprite);
```

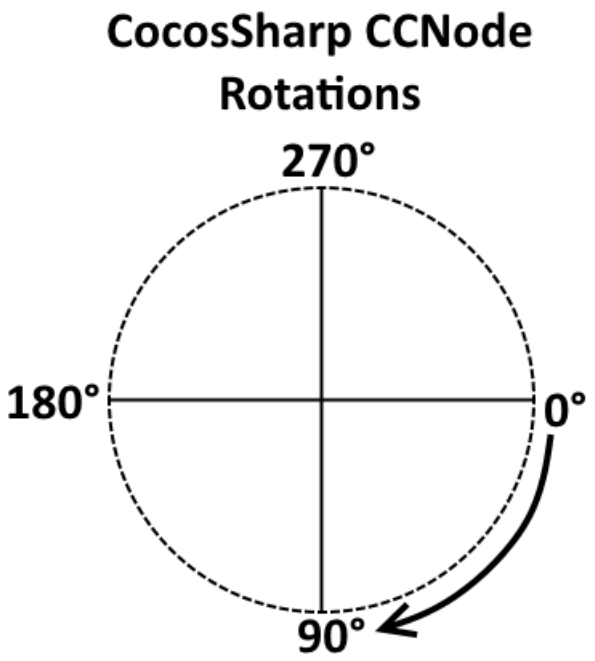
This results in the following:

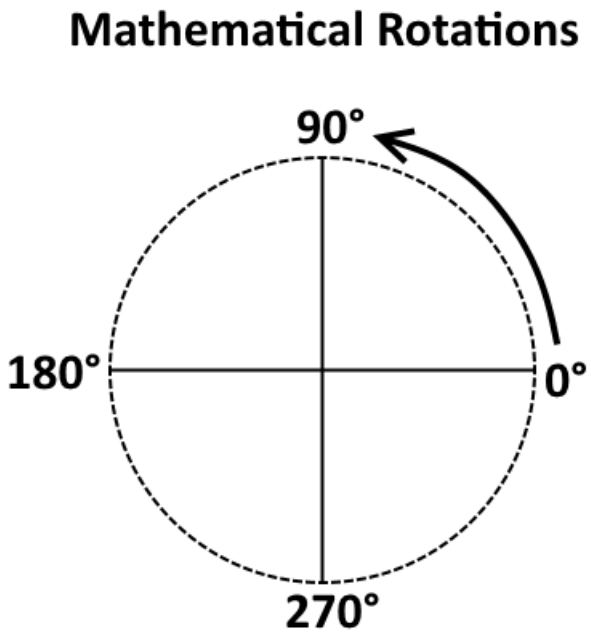


Notice that the rotation is 45 degrees clockwise (which for historical reasons is the opposite of how rotation is applied mathematically):



In general rotation for CocosSharp and regular mathematics can be visualized as follows:





This distinction is important because the `System.Math` class uses counterclockwise rotation, so developers familiar with this class need to invert angles when working with `CCNode` instances.

We should note that the above diagrams display rotation in degrees; however, some mathematical functions (such as the functions in the `System.Math` namespace) expect and return values in *radians* rather than degrees. We'll look at how to convert between the two unit types a little later in this guide.

Rotating to face a direction

As shown above, `CCSprite` can be rotated using the `Rotation` property. The `Rotation` property is provided by `CCNode` (the base class for `CCSprite`), which means rotation can be applied to entities which inherit from `CCNode` as well.

Some games require objects to be rotated so they face a target. Examples include a computer-controlled enemy shooting at a player target, or a space ship flying towards the point where the user is touching the screen. However, a rotation value must first be calculated based on the location of the entity being rotated and the location of the target to face.

This process requires a number of steps:

- Identifying the *offset* of the target. Offset refers to the X and Y distance between the rotating entity and the target entity.
- Calculating the angle from the offset by using the arctangent trigonometry function (explained in detail below).
- Adjusting for a difference between 0 degrees pointing towards the right and the direction that the rotating entity points when un-rotated.
- Adjusting for the difference between mathematical rotation (counterclockwise) and CocosSharp rotation (clockwise).

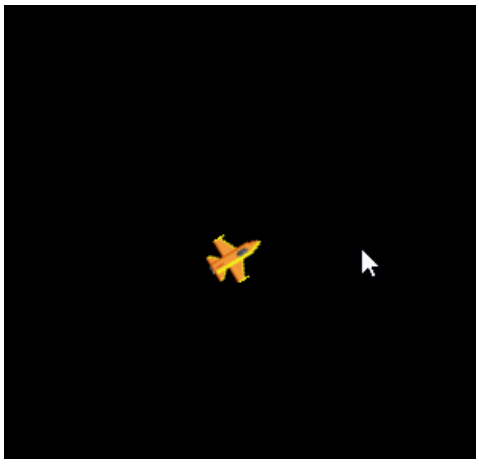
The following function (assumed to be written in an entity) rotates the entity to face a target:

```
// This function assumes that it is contained in a CCNode-inheriting object
public void FacePoint(float targetX, float targetY)
{
    // Calculate the offset - the target's position relative to "this"
    float xOffset = targetX - this.PositionX;
    float yOffset = targetY - this.PositionY;

    // Make sure the target isn't the same point as "this". If so,
    // then rotation cannot be calculated.
    if (targetX != this.PositionX || targetY != this.Position.Y)
    {

        // Call Atan2 to get the radians representing the angle from
        // "this" to the target
        float radiansToTarget = (float)System.Math.Atan2 (yOffset, xOffset);

        // Since CCNode uses degrees for its rotation, we need to convert
        // from radians
        float degreesToTarget = CCMathHelper.ToDegrees (radiansToTarget);

        // The direction that the entity faces when unrotated. In this case
        // the entity is facing "up", which is 90 degrees
        const float forwardAngle = 90;

        // Adjust the angle we want to rotate by subtracting the
        // forward angle.
        float adjustedForDirecitonFacing = degreesToTarget - forwardAngle;

        // Invert the angle since CocosSharp uses clockwise rotation
        float cocosSharpAngle = adjustedForDirecitonFacing * -1;

        // Finally assign the rotation
        this.Rotation = rotation = cocosSharpAngle;
    }
}
```

The code above could be used to rotate an entity so it faces the point where the user is touching the screen, as follows:

```
private void HandleInput(System.Collections.Generic.List<CCTouch> touches, CCEvent touchEvent)
{
    if(touches.Count > 0)
    {
        CCTouch firstTouch = touches[0];
        FacePoint (firstTouch.Location.X, firstTouch.Location.Y);
    }
}
```

This code results in the following behavior:



Using Atan2 to convert offsets to angles

`System.Math.Atan2` can be used to convert an offset to an angle. The function name `Atan2` comes from the trigonometric function arctangent. The "2" suffix differentiates this function from the standard `Atan` function, which strictly matches the mathematical behavior of arctangent. Arctangent is a function which returns a value between -90 and +90 degrees (or the equivalent in radians). Many applications, including computer games, often require a full 360 degrees of values, so the `Math` class includes `Atan2` to satisfy this need.

Notice that the code above passes the Y parameter first, then the X parameter, when calling the `Atan2` method. This is backwards from the usual X, Y ordering of position coordinates. For more information see the Atan2 docs.

It's also worth noting that the return value from `Atan2` is in radians, which is another unit used to measure angles. This guide doesn't cover the details of radians, but keep in mind that all trigonometric functions in the `System.Math` namespace use radians, so any values must be converted to degrees before being used on CocosSharp objects. More information on radians can be found in the radian Wikipedia page.

Forward angle

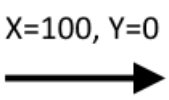
Once the `FacePoint` method converts the angle to radians, it defines a `forwardAngle` value. This value represents the angle in which the entity is facing when its Rotation value equals 0. In this example, we assume that the entity is facing upward, which is 90 degrees when using a mathematical rotation (as opposed to CocosSharp rotation). We use the mathematical rotation here since we haven't yet inverted the rotation for CocosSharp.

The following shows what an entity with a `forwardAngle` of 90 degrees might look like:

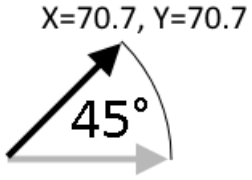
Angled velocity

So far we've looked at how to convert an offset into an angle. This section goes the other way – takes an angle and converts it into X and Y values. Common examples include a car moving in the direction that it is facing, or a space ship shooting a bullet which moves in the direction that the ship is facing.

Conceptually, velocity can be calculated by first defining the desired velocity when un-rotated, then rotating that velocity to the angle that an entity is facing. To help explain this concept, velocity (and acceleration) can be visualized as a 2-dimensional *vector* (which is typically drawn as an arrow). A vector for a velocity value with X = 100 and Y = 0 can be visualized as follows:



This vector can be rotated to result in a new velocity. For example, rotating the vector by 45 degrees (using counter-clockwise rotation) results in the following:



Fortunately, velocity, acceleration, and even position can all be rotated with the same code regardless of how the values are applied. The following general-purpose function can be used to rotate a vector by a CocosSharp Rotation value:

```
// Rotates the argument vector by degrees specified by
// cocosSharpDegrees. In other words, the rotation
// value is expected to be clockwise.
// The vector parameter is modified, so it is both an in and out value
void RotateVector(ref CCVector2 vector, float cocosSharpDegrees)
{
    // Invert the rotation to get degrees as is normally
    // used in math (counterclockwise)
    float mathDegrees = -cocosSharpDegrees;

    // Convert the degrees to radians, as the System.Math
    // object expects arguments in radians
    float radians = CCMathHelper.ToRadians (mathDegrees);

    // Calculate the "up" and "right" vectors. This is essentially
    // a 2x2 matrix that we'll use to rotate the vector
    float xAxisXComponent = (float)System.Math.Cos (radians);
    float xAxisYComponent = (float)System.Math.Sin (radians);
    float yAxisXComponent = (float)System.Math.Cos (radians + CCMathHelper.Pi / 2.0f);
    float yAxisYComponent = (float)System.Math.Sin (radians + CCMathHelper.Pi / 2.0f);

    // Store the original vector values which will be used
    // below to perform the final operation of rotation.
    float originalX = vector.X;
    float originalY = vector.Y;

    // Use the axis values calculated above (the matrix values)
    // to rotate and assign the vector.
    vector.X = originalX * xAxisXComponent + originalY * yAxisXComponent;
    vector.Y = originalX * xAxisYComponent + originalY * yAxisYComponent;
}
```

A full understanding of the `RotateVector` method requires being familiar with the cosine and sine trigonometric functions along with some linear algebra, which is beyond the scope of this article. However, once implemented the `RotateVector` method can be used to rotate any vector, including a velocity vector. For example, the following code may fire a bullet in a direction specified by the `rotation` value:

```
// Create a Bullet instance
Bullet newBullet = new Bullet();

// Define the velocity of the bullet when
// rotation is 0
CCVector2 velocity = new CCVector2 (0, 100);

// Modify the velocity according to rotation
RotateVector (ref velocity, rotation);

// Assign the newBullet's velocity using the
// rotated vector
newBullet.VelocityX = velocity.X;
newBullet.VelocityY = velocity.Y;

// Set the bullet's rotation so it faces
// the direction that it's flying
newBullet.Rotation = rotation;
```

This code may produce something like:

Summary

This guide covers common mathematical concepts in 2D game development. It shows how to assign and implement velocity and acceleration, and covers how to rotate objects and vectors for movement in any direction.

Performance and visual effects with CCRenderTexture

11/11/2018 • 11 minutes to read • Edit Online

CCRenderTexture enables developers to improve the performance of their CocosSharp games by reducing draw calls, and can be used for creating visual effects. This guide accompanies the CCRenderTexture sample to provide a hands-on example of how to use this class effectively.

The `CCRenderTexture` class provides functionality for rendering multiple CocosSharp objects to a single texture. Once created, `CCRenderTexture` instances can be used to render graphics efficiently and to implement visual effects. `CCRenderTexture` allows multiple objects to be rendered to a single texture one time. Then, that texture can be reused every frame, reducing the total number of draw calls.

This guide examines how to use the `CCRenderTexture` object to improve the performance of rendering cards in a collectable card game (CCG). It also demonstrates how `CCRenderTexture` can be used to make an entire entity transparent. This guide references the `CCRenderTexture` sample project.

Card – a typical entity

Before looking at how to use `CCRenderTexture` object, we'll first familiarize ourselves with the `Card` entity that we'll use throughout this project to explore the `CCRenderTexture` class. The `Card` class is a typical entity, following the entity pattern outlined in the Entity guide. The Card class has all of its visual components (instances of `CCSprite` and `CCLabel`) listed as fields:

```
class Card : CCNode
{
    bool usesRenderTexture;
    List<CCNode> visualComponents = new List<CCNode>();
    CCSprite background;
    CCSprite colorIcon;
    CCSprite monsterSprite;
    CCLabel monsterNameDisplay;
    CCLabel hpDisplay;
    CCLabel descriptionDisplay;
```

Card instances can be rendered by using a `CCRenderTexture`, or by drawing each visual component individually. Although each component is an independent object, the `CCNode` parenting system used in entities makes the `Card` behave as a single object – at least for the most part. For example, if a `Card` entity is repositioned, resized, or rotated, then all the contained visual objects are impacted to make the card appear to be a single object. To see the cards behave as a single object, we can modify the `GameLayer.AddedToScene` method to set the `useRenderTextures` variable to `false`:

```
protected override void AddedToScene ()
{
    base.AddedToScene();
    GameView.Stats.Enabled = true;
    const bool useRenderTextures = false;
    ...
}
```

The `GameLayer` code does not move each visual element independently, yet each visual element within the `Card` entity is positioned correctly:

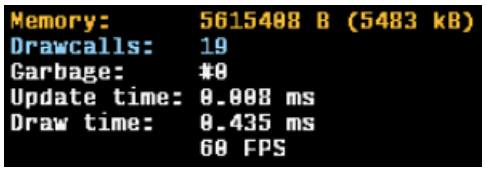
The sample is coded to expose two problems that can occur when each visual component renders itself:

- Performance can suffer due to multiple draw calls
- Certain visual effects, such as transparency, cannot be implemented accurately, as we will explore later

Card draw calls

Our code is a simplification of what might be found in a full *collectable card game* (CCG) such as "Magic: The Gathering" or "Hearthstone". Our game only displays three cards at once and has a small number of possible units (blue, green, and orange). By contrast, a full game may have over twenty cards on-screen at a given time, and players may have hundreds of cards to choose from when creating their decks. Even though our game does not currently suffer from performance problems, a full game with similar implementation might.

CocosSharp provides some insight into rendering performance by exposing the draw calls performed per-frame. Our `GameLayer.AddedToScene` method sets the `GameView.Stats.Enabled` to `true`, resulting in performance information shown at the bottom-left of the screen:



Notice that despite having three cards on screen, we have nineteen draw calls (each card results in six draw calls, the text displaying the performance information accounts for one more). Draw calls have a significant impact on a game's performance, so CocosSharp provides a number of ways to reduce them. One technique is described in the CCSpriteSheet guide. Another technique is to use the `CCRenderTexture` to reduce each entity down to one call, as we'll examine in this guide.

Card transparency

Our `Card` entity includes an `Opacity` property to control transparency as shown in the following code snippet:

```
public override byte Opacity
{
    get
    {
        return base.Opacity;
    }
    set
    {
        base.Opacity = value;
        if (usesRenderTexture)
        {
            this.renderTexture.Sprite.Opacity = value;
        }
        else
        {
            foreach (var component in visualComponents)
            {
                component.Opacity = value;
            }
        }
    }
}
```

Notice that the setter supports using render textures or rendering each component individually. To see its effect, change the `opacity` value to `127` (roughly half opacity) in `GameLayer.AddedToScene` which will result in each component having an `Opacity` value of `127`:

```
protected override void AddedToScene ()
{
    base.AddedToScene();
    GameView.Stats.Enabled = true;
    const bool useRenderTextures = false;
    const byte opacity = 127;
    ...
}
```

The game will now render the cards with some transparency, causing them to appear darker since the background is black:

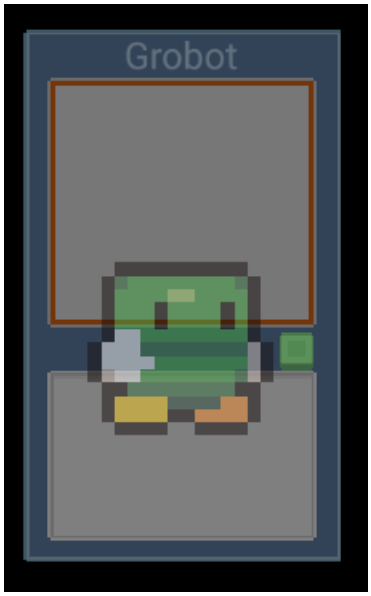
At first glance it might look as if our cards have been properly made transparent. However, the screenshot displays a number of visual problems.

Since our background is black, we would expect that every part of our card would become darker due to the transparency. That is, the more transparent a card becomes, the darker it becomes. At opacity 0, a `Card` will be completely transparent (totally black). However, some parts of our card didn't become darker when opacity was changed to `127`. Even worse, some parts of our card actually became brighter when they became more transparent. Let's look at parts of our card which were black *before* they were transparent – specifically the detail text and the black outlines around the monster graphic. If we place these side by side, we can see the impact of applying transparency:

As mentioned above, all parts of the card should become darker when becoming more transparent, but in a number of areas this is not the case:

- The robot's outline becomes lighter (goes from black to gray)
- The description text becomes lighter (goes from black to gray)
- The green part of the robot becomes less saturated, but doesn't become darker

To help visualize why this occurs, we need to keep in mind that each visual component is drawn independently, each partly transparent. The first visual component drawn is the card's background. Subsequent transparent elements will be drawn on top of the card and will be impacted by the card background. If we remove some text from our card and move the robot graphic down, we can see how the card background impacts the robot. Notice the orange line from the top box can be seen on the robot, and that the area of the robot which overlaps the blue stripe in the center of the card is drawn darker:



Using a `CCRenderTexture` allows us to make the entire card transparent without impacting the rendering of individual components within the card, as we will see later in this guide.

Using CCRenderTexture

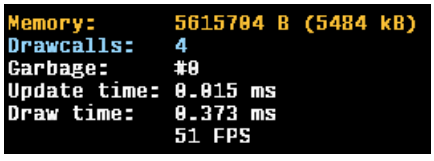
Now that we've identified the problems with rendering each component individually, we'll turn on rendering to a `CCRenderTexture` and compare the behavior.

To enable rendering to a `CCRenderTexture`, change the `userRenderTextures` variable to `true` in `GameLayer.AddedToScene`:

```
protected override void AddedToScene ()
{
    base.AddedToScene();
    GameView.Stats.Enabled = true;
    const bool useRenderTextures = true;
```

Card draw calls

If we run the game now, we'll see the draw calls reduced from nineteen to four (each card reduced from six to one):



As previously mentioned, this type of reduction can have a significant impact on games with more visual entities on screen.

Card transparency

Once the `useRenderTextures` is set to `true`, transparent cards will render differently:

Let's compare the transparent robot card using render textures (left) vs. without (right):

The most obvious differences are in the details text (black instead of light gray) and the robot sprite (dark instead of light and desaturated).

CCRenderTexture details

Now that we've seen the benefits of using `CCRenderTexture`, let's take a look at how it is used in the `Card` entity.

The `CCRenderTexture` is a canvas that can be the target of rendering. It has two main differences when compared to the game screen:

1. The `CCRenderTexture` persists in-between frames. This means that a `CCRenderTexture` needs to only be rendered when changes occur. In our case, the `Card` entity never changes, so it is only rendered one time. If any `Card` components changed, then the Card would need to redraw itself to its `CCRenderTexture`. For example, if the HP value (health points) changed when attacked, then the card would need to render itself to reflect the new HP value.
2. The `CCRenderTexture` pixel dimensions are not tied to the screen. A `CCRenderTexture` can be larger or smaller

than the resolution of the device. The `Card` code creates a `CCRenderTexture` using the size of its background sprite. The card contains a reference to a `CCRenderTexture` called `renderTexture`:

```
CCRenderTexture renderTexture;
```

The `renderTexture` instance remains `null` until the `UseRenderTexture` property is assigned to true, which calls `SwitchToRenderTexture`:

```
private void SwitchToRenderTexture()
{
    // The card needs to be moved to the origin (0,0) so it's rendered on the render target.
    // After it's rendered to the CCRenderTexture, it will be moved back to its old position
    var oldPosition = this.Position;
    // Make sure visuals are part of the card so they get rendered
    bool areVisualComponentsAlreadyAdded = this.Children != null &&
this.Children.Contains(visualComponents[0]);
    if (!areVisualComponentsAlreadyAdded)
    {
        // Temporarily add them so we can render the object:
        foreach (var component in visualComponents)
        {
            this.AddChild(component);
        }
    }
    // Create the render texture if it hasn't yet been made:
    if (renderTexture == null)
    {
        // Even though the game is zoomed in to create a pixellated look, we are using
        // high-resolution textures. Therefore, we want to have our canvas be 2x as big as
        // the background so fonts don't appear pixellated
        var pixelResolution = background.ContentSize * 2;
        var unitResolution = background.ContentSize;
        renderTexture = new CCRenderTexture(unitResolution, pixelResolution);
        //renderTexture.Sprite.Scale = .5f;
    }
    // We don't want the render target to be a child of this when we update it:
    if (this.Children != null && this.Children.Contains(renderTexture.Sprite))
    {
        this.Children.Remove(renderTexture.Sprite);
    }
    // Move this instance back to the origin so it is rendered inside the render texture:
    this.Position = CCPoint.Zero;
    // Clears the CCRenderTexture
    renderTexture.BeginWithClear(CCColor4B.Transparent);
    // Visit renders this object and all of its children
    this.Visit();
    // Ends the rendering, which means the CCRenderTexture's Sprite can be used
    renderTexture.End();
    // We no longer want the individual components to be drawn, so remove them:
    foreach (var component in visualComponents)
    {
        this.RemoveChild(component);
    }
    // Move this back to its original position:
    this.Position = oldPosition;
    // add the render texture sprite to this:
    renderTexture.Sprite.AnchorPoint = CCPoint.Zero;
    this.AddChild(renderTexture.Sprite);
}
```

The `SwitchToRenderTexture` method can be called whenever the texture needs to be refreshed. It can be called whether the card is already using its `CCRenderTexture` or is switching to the `CCRenderTexture` for the first time.

The following sections explore the `SwitchToRenderTexture` method.

CCRenderTexture size

The CCRenderTexture constructor requires two sets of dimensions. The first controls the size of the `CCRenderTexture` when it is drawn, and the second specifies the pixel width and height of its contents. The `Card` entity instantiates its `CCRenderTexture` using the background ContentSize. Our game has a `DesignResolution` of 512 by 384, as shown in `ViewController.LoadGame` on iOS and `MainActivity.LoadGame` on Android:

```
int width = 512;
int height = 384;
// Set world dimensions
gameView.DesignResolution = new CCSizeI(width, height);
```

The `CCRenderTexture` constructor is called with the `background.ContentSize` as the first parameter, indicating that the `CCRenderTexture` should be just as large as the background `CCSprite`. Since the card background `CCSprite` is 200 pixels tall, the card will occupy roughly half of the vertical height of the screen.

The second parameter passed to the `CCRenderTexture` constructor specifies the pixel resolution of the `CCRenderTexture`. As discussed in the CocosSharp Resolution guide, the width and height of the viewable area in game units is often not the same as the pixel resolution of the screen. Similarly, a CCRenderTexture might use a larger resolution than its size so visuals appear crisper on high-resolution devices.

The pixel resolution is twice the size of the CCRenderTexture to prevent text from looking pixelated:

```
var unitResolution = background.ContentSize;
var pixelResolution = background.ContentSize * 2;
renderTexture = new CCRenderTexture(unitResolution, pixelResolution);
```

To compare, we can change the pixelResolution value to match the `background.ContentSize` (without being doubled) and compare the result:

```
var unitResolution = background.ContentSize;
var pixelResolution = background.ContentSize;
renderTexture = new CCRenderTexture(unitResolution, pixelResolution);
```

Rendering to a CCRenderTexture

Typically, visual objects in CocosSharp are not explicitly rendered. Instead, visual objects are added to a `CCLayer` which is part of a `CCScene`. CocosSharp automatically renders the `CCScene` and its visual hierarchy in every frame without any rendering code being called.

By contrast, the `CCRenderTexture` must be explicitly drawn to. This rendering can be broken up into three steps:

1. `CCRenderTexture.BeginWithClear` is called, indicating that all subsequent rendering will target the calling `CCRenderTexture`.
2. Objects inheriting from `CCNode` (like the `Card` entity) are rendered to the `CCRenderTexture` by calling `Visit`.
3. `CCRenderTexture.End` is called, indicating that rendering to the `CCRenderTexture` is complete.

Any number of objects can be rendered to a `CCRenderTexture` between its `Begin` and `End` calls. Before rendering, all necessary visible objects are added as children:

```
bool areVisualComponentsAlreadyAdded = this.Children != null && this.Children.Contains(visualComponents[0]);
if (!areVisualComponentsAlreadyAdded)
{
    // Temporarily add them so we can render the object:
    foreach (var component in visualComponents)
    {
        this.AddChild(component);
    }
}
```

The `renderTexture` should not be part of the card when rendering, so it is removed:

```
// We don't want the render texture to be a child of the card
// when we call Visit
if (this.Children != null && this.Children.Contains(renderTexture.Sprite))
{
    this.RemoveChild(renderTexture.Sprite);
}
```

Now the `Card` instance can render itself to the `CCRenderTexture` instance:

```
// Clears the CCRenderTexture
renderTexture.BeginWithClear(CCColor4B.Transparent);
// Visit renders this object and all of its children
this.Visit();
// Ends the rendering, which means the CCRenderTexture's Sprite can be used
renderTexture.End();
```

Once the rendering is finished, the individual components are removed and the `CCRenderTexture` is re-added.

```
// We no longer want the individual components to be drawn, so remove them:
foreach (var component in visualComponents)
{
    this.RemoveChild(component);
}
// add the render target sprite to this:
this.AddChild(renderTexture.Sprite);
```

Summary

This guide covered the `CCRenderTexture` class by using a `Card` entity which could be used in a collectible card game. It showed how to use the `CCRenderTexture` class to improve frame rate and properly implement entity-wide

transparency.

Although this guide used a `CCRenderTexture` contained within an entity, this class can be used to render multiple entities, or even entire `CCLayer` instances for screen-wide effects and performance improvements.

Related Links

- CCRenderTexture API Reference
- Full Project (sample)

Monogame Framework

10/3/2018 • 2 minutes to read • Edit Online

MonoGame is an efficient, flexible, cross-platform API for developing 2D and 3D games. It provides the foundation for many cross-platform game engines, but can be used directly in games without being wrapped in a game engine.

Introduction to Game Development with MonoGame

MonoGame is a cross-platform, hardware accelerated API providing graphics, audio, game state management, input, and a content pipeline for importing assets.

3D Graphics with MonoGame

MonoGame offers a flexible, efficient API for displaying real-time 3D graphics. It includes higher-level constructs for rendering and also access to lower-level graphics resources.

MonoGame GamePad Reference

GamePad is a standard, cross-platform class for accessing input devices in MonoGame.

MonoGame Platform Specific Considerations

MonoGame is supported on a variety of platforms. This section covers topics specific to each platform.

Introduction to Game Development with MonoGame

10/3/2018 • 2 minutes to read • Edit Online

This multi-part walkthrough shows how to create a simple 2D application using MonoGame. It covers common game programming concepts, such as graphics, input, game entities, and physics.

This article describes MonoGame API technology for making cross-platform games. For a full list of platforms, see the MonoGame website. This tutorial will use C# for code samples, although MonoGame is fully functional with F# as well.

MonoGame is a cross-platform, hardware accelerated API providing graphics, audio, game state management, input, and a content pipeline for importing assets. Unlike most game engines, MonoGame does not provide or impose any pattern or project structure. While this means that developers are free to organize their code as they like, it also means that a bit of setup code is needed when first starting a new project.

The first section of this walkthrough focuses on setting up an empty project. The last section covers writing all of our game logic and content – most of which will be cross platform.

By the end of this walkthrough, we will have created a simple game where the player can control an animated character with touch input. Although this is not technically a full game (since it has no win or lose conditions), it demonstrates numerous game development concepts and can be used as the foundation for many types of games.

The following shows the result of this walkthrough:

Monogame and XNA

The MonoGame library is intended to mimic Microsoft's XNA library in both syntax and functionality. All MonoGame objects exist under the Microsoft.Xna namespace – allowing most XNA code to be used in MonoGame with no modification.

Developers familiar with XNA will already be familiar with MonoGame's syntax, and developers looking for additional information on working with MonoGame will be able to reference existing online XNA walkthroughs, API documentation, and discussions.

Walkthrough Parts

- Part 1 – Creating a Cross Platform MonoGame Project

- Part 2 – Implementing the WalkingGame

Related Links

- WalkingGame MonoGame Project (sample)
- XNB Fonts iOS
- XNB Fonts Android
- MonoGame Android on NuGet
- MonoGame iOS on NuGet
- MonoGame API Documentation

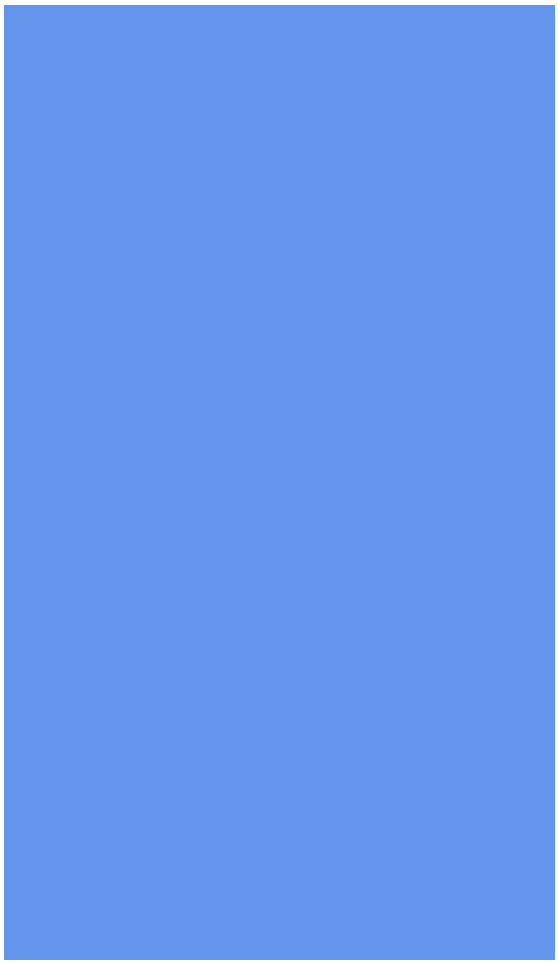
Part 1 – Creating a Cross Platform MonoGame

10/3/2018 • 2 minutes to read • Edit Online

This walkthrough shows how to create a new project for iOS and Android using MonoGame. The result is a Visual Studio for Mac solution with a cross-platform shared code project as well as one project for each platform. This project will display an empty blue screen when executed.

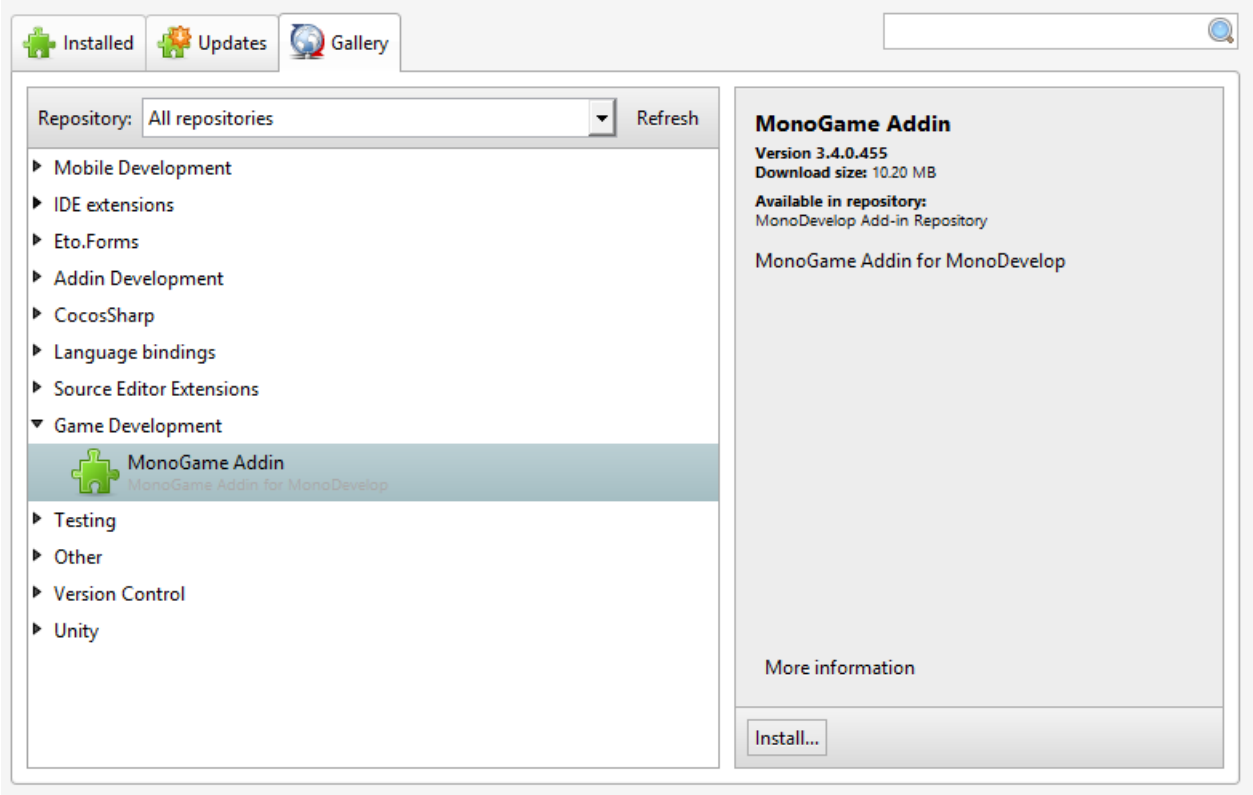
MonoGame enables the development of cross-platform games with large portion of code reuse. This walkthrough will focus on setting up a solution which contains projects for iOS and Android, as well as a shared code project for cross-platform code.

When we're done, we'll have a project that has the proper structure for performing game update logic and game drawing logic at 30 frames per second. It can be used as the base project for any MonoGame project. Our project will look like this when executed:



Adding MonoGame to Visual Studio for Mac

MonoGame can be added as an add-in to Visual Studio for Mac. On Mac, select **Visual Studio for Mac** > **Add-in Manager...** . On Windows, select** Tools **> **Add-in Manager...** . Select the **Gallery** tab, expand the **Game Development** category and select **MonoGame Addin**, then click **Install**:



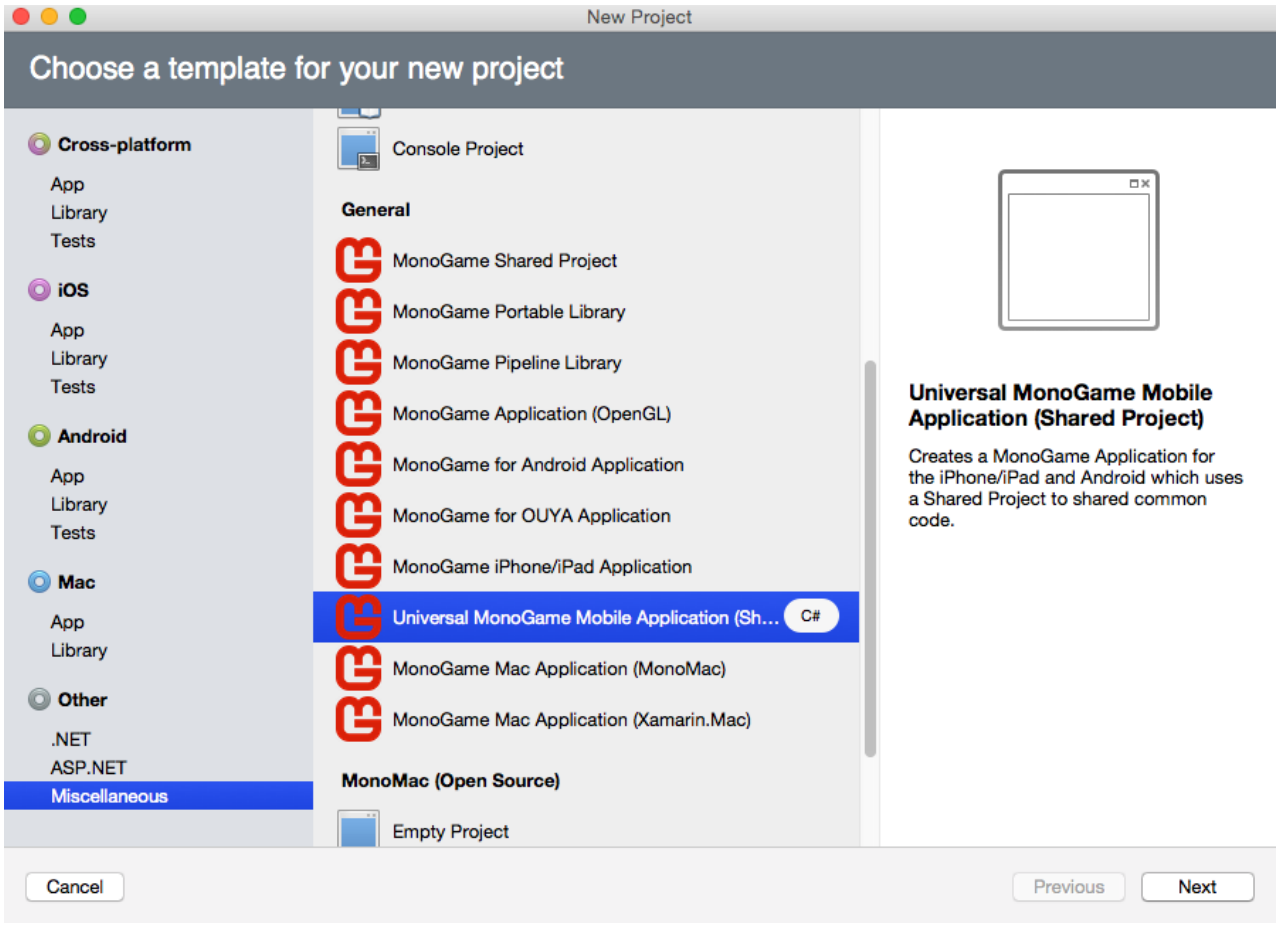
IMPORTANT

Note: If the **Game Development** section does not appear in the Add-in Manager, you can manually download and install the latest version from here: http://www.monogame.net/downloads/. You may need to restart Visual Studio for Mac for the templates to appear.

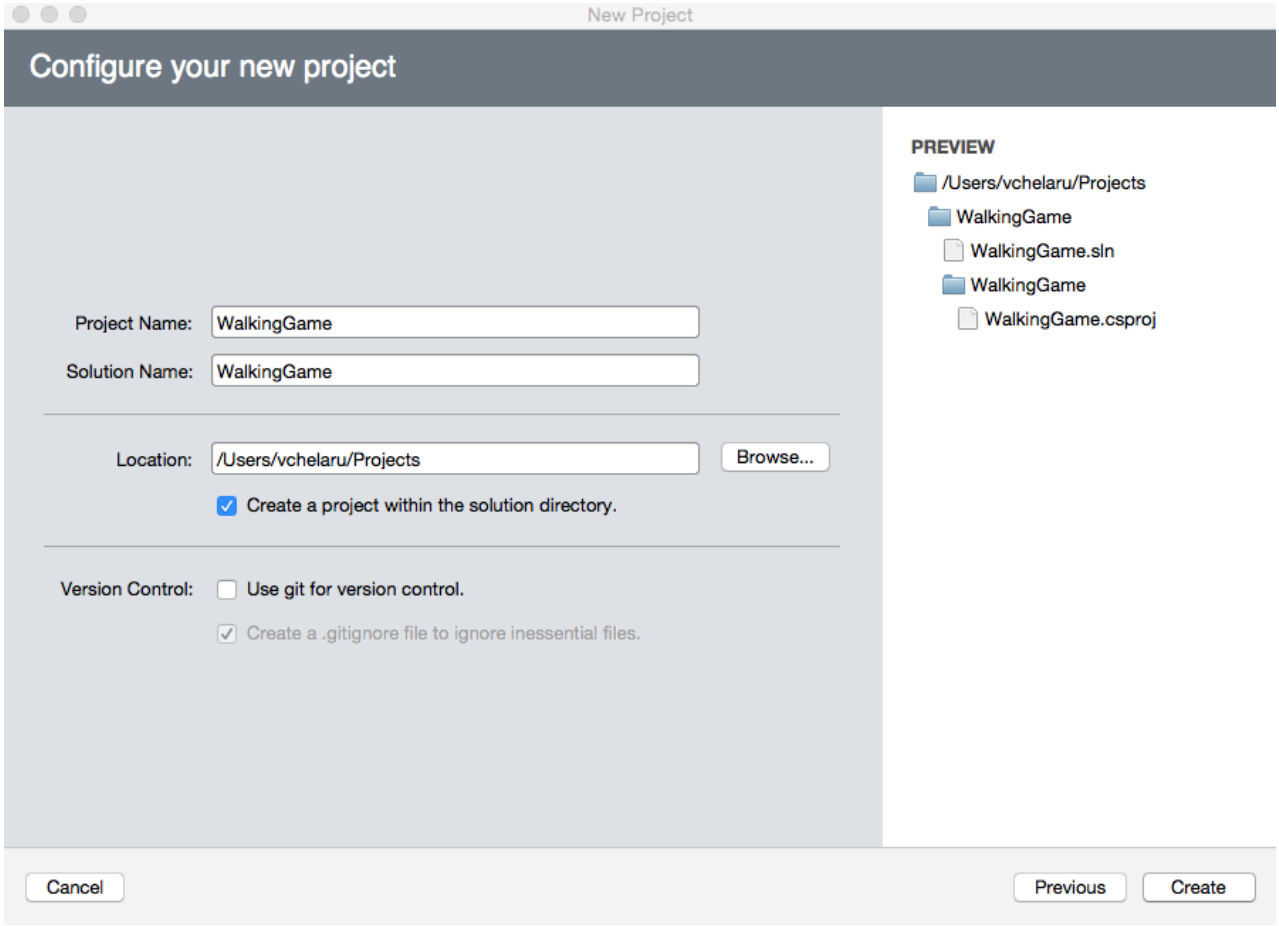
Once installed, MonoGame templates will appear in Visual Studio for Mac, as we will see in the next section.

Creating a new solution

In Visual Studio for Mac select **File > New Solution**. In the **New Project** dialog, click on **Miscellaneous**, scroll to the **General** section, select the **Universal MonoGame Mobile application **option, and click Next.



Name the project WalkingGame and click Create:



Now our project will execute just like any other iOS or Android project. The project should run displaying a cornflower blue background:

Fixing Android Compile Errors

The current version of MonoGame's templates includes a few syntax errors in the Android's `Activity1.cs` file. To fix these problems, replace the `OnCreate` function with the following:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    var g = new Game1();
    SetContentView((View)g.Services.GetService(typeof(View)));
    g.Run();
}
```

Summary

This walkthrough covered how to create a cross-platform MonoGame project using Visual Studio for Mac. The result of this is an empty blue screen. This project can be used as the starting point for any iOS and Android game.

Related Links

- MonoGame Android NuGet
- MonoGame iOS NuGet
- MonoGame API Documentation

Part 2 – Implementing the WalkingGame

1/23/2019 • 18 minutes to read • Edit Online

This walkthrough shows how to add game logic and content to an empty MonoGame project to create a demo of an animated sprite moving with touch input.

The previous parts of this walkthrough showed how to create empty MonoGame projects. We will build on these previous parts by making a simple game demo. This article contains the following sections:

- Unzipping our game content
- MonoGame Class Overview
- Rendering our first Sprite
- Creating the CharacterEntity
- Adding CharacterEntity to the game
- Creating the Animation class
- Adding the first Animation to CharacterEntity
- Adding movement to the character
- Matching movement and animation

Unzipping our Game Content

Before we begin writing code, we will want to unzip our game *content*. Game developers often use the term *content* to refer to non-code files which are usually created by visual artists, game designers, or audio designers. Common types of content include files used to display visuals, play sound, or control artificial intelligence (AI) behavior. From a game development team's perspective content is usually created by non-programmers.

The content used here can be found on github. We'll need these files downloaded to a location that we will access later in this walkthrough.

MonoGame Class Overview

For starters we will explore the MonoGame classes used in basic rendering:

- `SpriteBatch` – used to draw 2D graphics to the screen. *Sprites* are 2D visual elements which are used to display images on screen. The `SpriteBatch` object can draw a single sprite at a time between its `Begin` and `End` methods, or multiple sprites can be grouped together, or *batched*.
- `Texture2D` – represents an image object at runtime. `Texture2D` instances are often created from file formats such as .png or .bmp, although they can also be created dynamically at runtime. `Texture2D` instances are used when rendering with `SpriteBatch` instances.
- `Vector2` – represents a position in a 2D coordinate system which is often used for positioning visual objects. MonoGame also includes `Vector3` and `Vector4` but we will only use `Vector2` in this walkthrough.
- `Rectangle` – a four-sided area with position, width, and height. We'll be using this to define which portion of our `Texture2D` to render when we start working with animations.

We should also note that MonoGame is not maintained by Microsoft – despite its namespace. The `Microsoft.Xna` namespace is used in MonoGame to make it easier to migrate existing XNA projects to MonoGame.

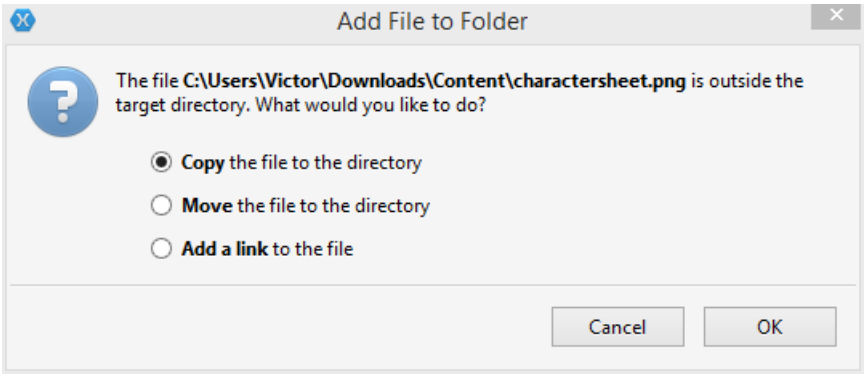
Rendering our First Sprite

Next we will draw a single sprite to the screen to show how to perform 2D rendering in MonoGame.

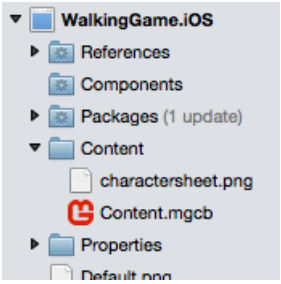
Creating a Texture2D

We need to create a `Texture2D` instance to use when rendering our sprite. All game content is ultimately contained in a folder named **Content,** located in the platform-specific project. MonoGame shared projects cannot contain content, as the content must use build actions specific to the platform. CocosSharp developers will find the Content folder a familiar concept – they are located in the same place in both CocosSharp and MonoGame projects. The Content folder can be found in the iOS project, and inside the Assets folder in the Android project.

To add our game's content, right-click on the **Content** folder and select **Add > Add Files...** Navigate to the location where the content.zip file was extracted and select the **charactersheet.png** file. If asked about how to add the file to folder, we should select the **Copy** option:



The Content folder now contains the charactersheet.png file:



Next, we'll add code to load the charactersheet.png file and create a `Texture2D`. To do this open the `Game1.cs` file and add the following field to the Game1.cs class:

```
Texture2D characterSheetTexture;
```

Next, we'll create the `characterSheetTexture` in the `LoadContent` method. Before any modifications `LoadContent` method looks like this:

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
    // TODO: use this.Content to load your game content here
}
```

We should note that the default project already instantiates the `spriteBatch` instance for us. We'll be using this later but we won't be adding any additional code to prepare the `spriteBatch` for use. On the other hand, our `spriteSheetTexture` does require initialization, so we will initialize it after the `spriteBatch` is created:

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
    using (var stream = TitleContainer.OpenStream ("Content/charactersheet.png"))
    {
        characterSheetTexture = Texture2D.FromStream (this.GraphicsDevice, stream);

    }
}
```

Now that we have a `SpriteBatch` instance and a `Texture2D` instance we can add our rendering code to the `Game1.Draw` method:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin ();

    Vector2 topLeftOfSprite = new Vector2 (50, 50);
    Color tintColor = Color.White;

    spriteBatch.Draw(characterSheetTexture, topLeftOfSprite, tintColor);

    spriteBatch.End ();

    base.Draw(gameTime);
}
```

Running the game now shows a single sprite displaying the texture created from charactersheet.png:

Creating the CharacterEntity

So far we've added code to render a single sprite to the screen; however, if we were to develop a full game without creating any other classes, we'd run into code organization issues.

What is an Entity?

A common pattern for organizing game code is to create a new class for each game *entity* type. An entity in game development is an object which can contain some of the following characteristics (not all are required):

- A visual element such as a sprite, text, or 3D model
- Physics or every frame behavior such as a unit patrolling a set path or a player character responding to input
- Can be created and destroyed dynamically, such as a power-up appearing and being collected by the player

Entity organization systems can be complex, and many game engines offer classes to help manage entities. We'll be implementing a very simple entity system, so it's worth noting that full games usually require more organization on the developer's part.

Defining the CharacterEntity

Our entity, which we'll call `CharacterEntity`, will have the following characteristics:

- The ability to load its own `Texture2D`
- The ability to render itself, including containing calling methods to update the walking animation
- `X` and Y properties to control the character's position.
- The ability to update itself – specifically, to read values from the touch screen and adjust position appropriately.

To add the `CharacterEntity` to our game, right-click or Control-click on the **WalkingGame** project and select **Add > New File...**. Select the **Empty Class** option and name the new file **CharacterEntity**, then click **New**.

First we'll add the ability for the `CharacterEntity` to load a `Texture2D` as well as to draw itself. We will modify the newly-added `CharacterEntity.cs` file as follows:

```
using System;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input.Touch;


namespace WalkingGame
{
    public class CharacterEntity
    {
        static Texture2D characterSheetTexture;

        public float X
        {
            get;
            set;
        }

        public float Y
        {
            get;
            set;
        }

        public CharacterEntity (GraphicsDevice graphicsDevice)
        {
            if (characterSheetTexture == null)
            {
                using (var stream = TitleContainer.OpenStream ("Content/charactersheet.png"))
                {
                    characterSheetTexture = Texture2D.FromStream (graphicsDevice, stream);
                }
            }
        }

        public void Draw(SpriteBatch spriteBatch)
        {
            Vector2 topLeftOfSprite = new Vector2 (this.X, this.Y);
            Color tintColor = Color.White;
            spriteBatch.Draw(characterSheetTexture, topLeftOfSprite, tintColor);
        }
    }
}
```

The above code adds the responsibility of positioning, rendering, and loading content to the `CharacterEntity`. Let's take a moment to point out some considerations taken in the code above.

Why are X and Y Floats?

Developers who are new to game programming may wonder why the `float` type is being used as opposed to `int` or `double`. The reason is that a 32-bit value is most common for positioning in low-level rendering code. The double type occupies 64 bits for precision, which is rarely needed for positioning objects. While a 32 bit difference may seem insignificant, many modern games include graphics which require processing tens of thousands of position values each frame (30 or 60 times per second). Cutting the amount of memory that must move through the graphics pipeline by half can have a significant impact on a game's performance.

The `int` data type can be an appropriate unit of measurement for positioning, but it can place limitations on the way entities are positioned. For example, using integer values makes it more difficult to implement smooth movement for games which support zooming in or 3D cameras (which are allowed by `SpriteBatch`).

Finally, we will see that the logic which moves our character around the screen will do so using the game's timing values. The result of multiplying velocity by how much time has passed in a given frame will rarely result in a whole number, so we need to use `float` for `X` and `Y`.

Why is characterSheetTexture Static?

The `characterSheetTexture` `Texture2D` instance is a runtime representation of the charactersheet.png file. In other words, it contains the color values for each pixel as extracted from the source **charactersheet.png** file. If our game were to create two `CharacterEntity` instances, then each one would need access to information from charactersheet.png. In this situation we wouldn't want to incur the performance cost of loading charactersheet.png twice, nor would we want to have duplicate texture memory stored in ram. Using a `static` field allows us to share the same `Texture2D` across all `CharacterEntity` instances.

Using `static` members for the runtime representation of content is a simplistic solution and it does not address problems encountered in larger games such as unloading content when moving from one level to another. More sophisticated solutions, which are beyond the scope of this walkthrough, include using MonoGame's content pipeline or creating a custom content management system.

Why is SpriteBatch Passed as an Argument Instead of Instantiated by the Entity?

The `Draw` method as implemented above takes a `SpriteBatch` argument, but by contrast, the `characterSheetTexture` is instantiated by the `CharacterEntity`.

The reason for this is because the most efficient rendering is possible when the same `SpriteBatch` instance is used for all `Draw` calls, and when all `Draw` calls are being made between a single set of `Begin` and `End` calls. Of course, our game will only include a single entity instance, but more complicated games will benefit from pattern that allows multiple entities to use the same `SpriteBatch` instance.

Adding CharacterEntity to the Game

Now that we've added our `CharacterEntity` with code for rendering itself, we can replace the code in `Game1.cs` to use an instance of this new entity. To do this we'll replace the `Texture2D` field with a `CharacterEntity` field in `Game1`:

```
public class Game1 : Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    // New code:
    CharacterEntity character;

    public Game1()
    {
    ...
```

Next, we'll add the instantiation of this entity in `Game1.Initialize`:

```
protected override void Initialize()
{
    character = new CharacterEntity (this.GraphicsDevice);

    base.Initialize();
}
```

We also need to remove the `Texture2D` creation from `LoadContent` since that is now handled inside of our `CharacterEntity`. `Game1.LoadContent` should look like this:

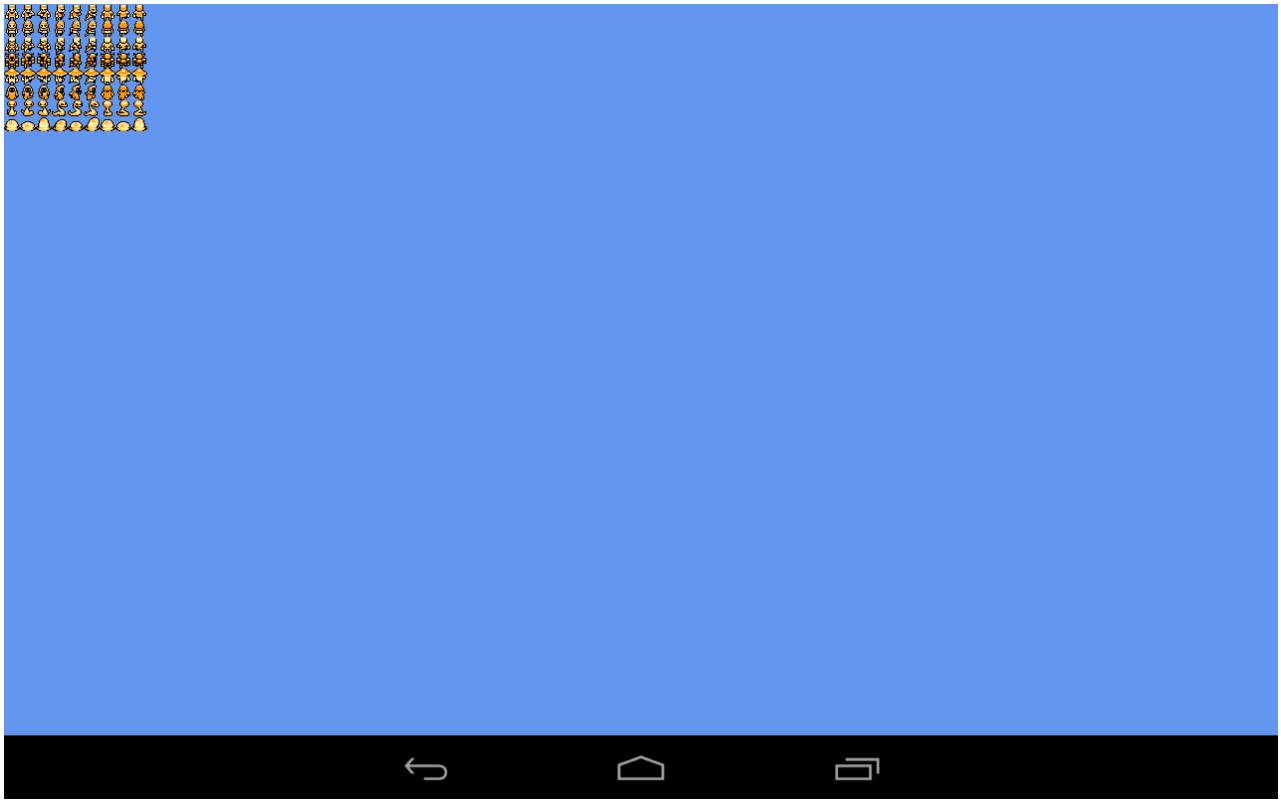
```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
}
```

It's worth noting that despite its name the `LoadContent` method is not the only place where content can be loaded – many games implement content loading in their Initialize method, or even in their Update code as content, may not be needed until the player reaches a certain point of the game.

Finally we can modify the Draw method as follows:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // We'll start all of our drawing here:
    spriteBatch.Begin ();

    // Now we can do any entity rendering:
    character.Draw(spriteBatch);
    // End renders all sprites to the screen:
    spriteBatch.End ();

    base.Draw(gameTime);
}
```

If we run the game, we will now see the character. Since X and Y default to 0, then the character is positioned against the top left corner of the screen:



Creating the Animation Class

Currently our `CharacterEntity` displays the full **charactersheet.png** file. This arrangement of multiple images in one file is referred to as a *sprite sheet*. Typically, a sprite will render only a portion of the sprite sheet. We will modify the `CharacterEntity` to render a portion of this **charactersheet.png**, and this portion will change over

time to display a walking animation.

We will create the `Animation` class to control the logic and state of the CharacterEntity animation. The Animation class will be a general class which could be used for any entity, not just `CharacterEntity` animations. Ultimate the `Animation` class will provide a `Rectangle` which the `CharacterEntity` will use when drawing itself. We'll also create an `AnimationFrame` class which will be used to define the animation.

Defining AnimationFrame

`AnimationFrame` will not contain any logic related to animation. We'll be using it only to store data. To add the `AnimationFrame` class, right-click or Control-click on the **WalkingGame** shared project and select **Add > New File...**. Enter the name **AnimationFrame** and click the **New** button. We'll modify the `AnimationFrame.cs` file so that it contains the following code:

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

namespace WalkingGame
{
    public class AnimationFrame
    {
        public Rectangle SourceRectangle { get; set; }
        public TimeSpan Duration { get; set; }
    }
}
```

The `AnimationFrame` class contains two pieces of information:

- `SourceRectangle` – Defines the area of the `Texture2D` which will be displayed by the `AnimationFrame`. This value is measured in pixels, with the top left being (0, 0).
- `Duration` – Defines how long an `AnimationFrame` is displayed when used in an `Animation`.

Defining Animation

The `Animation` class will contain a `List<AnimationFrame>` as well as the logic to switch which frame is currently displayed according to how much time has passed.

To add the `Animation` class, right-click or Control-click on the **WalkingGame** shared project and select **Add > New File...**. Enter the name **Animation** and click the **New** button. We'll modify the `Animation.cs` file so it contains the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;

namespace WalkingGame
{
    public class Animation
    {
        List<AnimationFrame> frames = new List<AnimationFrame>();
        TimeSpan timeIntoAnimation;

        TimeSpan Duration
        {
            get
            {
                double totalSeconds = 0;
                foreach (var frame in frames)
                {
                    totalSeconds += frame.Duration.TotalSeconds;
                }

                return TimeSpan.FromSeconds (totalSeconds);
            }
        }

        public void AddFrame(Rectangle rectangle, TimeSpan duration)
        {
            AnimationFrame newFrame = new AnimationFrame()
            {
                SourceRectangle = rectangle,
                Duration = duration
            };

            frames.Add(newFrame);
        }

        public void Update(GameTime gameTime)
        {
            double secondsIntoAnimation =
                timeIntoAnimation.TotalSeconds + gameTime.ElapsedGameTime.TotalSeconds;


            double remainder = secondsIntoAnimation % Duration.TotalSeconds;

            timeIntoAnimation = TimeSpan.FromSeconds (remainder);
        }
    }
}
```

Let's look at some of the details from the `Animation` class.

frames List

The `frames` member is what stores the data for our animation. The code which instantiates the animations will add `AnimationFrame` instances to the `frames` list through the `AddFrame` method. A more complete implementation may offer `public` methods or properties for modifying `frames`, but we'll limit the functionality to adding frames for this walkthrough.

Duration

Duration returns the total duration of the `Animation`, which is obtained by adding the duration of all of the contained `AnimationFrame` instances. This value could be cached if `AnimationFrame` were an immutable object, but since we implemented AnimationFrame as a class which can be changed after being added to Animation, we need to calculate this value whenever the property is accessed.

Update

The `Update` method should be called every frame (that is, every time the entire game is updated). Its purpose is to increase the `timeIntoAnimation` member which is used to return the currently displayed frame. The logic in `Update` prevents the `timeIntoAnimation` from ever being larger than the duration of the entire animation.

Since we'll be using the `Animation` class to display a walking animation then we want to have this animation repeat when it has reached the end. We can accomplish this by checking if the `timeIntoAnimation` is larger than the `Duration` value. If so it will cycle back to the beginning and preserve the remainder, resulting in a looping animation.

Obtaining the Current Frame's Rectangle

The purpose of the `Animation` class is ultimately to provide a `Rectangle` which can be used when drawing a sprite. Currently the `Animation` class contains logic for changing the `timeIntoAnimation` member, which we'll use to obtain which `Rectangle` should be displayed. We'll do this by creating a `CurrentRectangle` property in the `Animation` class. Copy this property into the `Animation` class:

```
public Rectangle CurrentRectangle
{
    get
    {
        AnimationFrame currentFrame = null;

        // See if we can find the frame
        TimeSpan accumulatedTime = new TimeSpan();
        foreach(var frame in frames)
        {
            if (accumulatedTime + frame.Duration >= timeIntoAnimation)
            {
                currentFrame = frame;
                break;
            }
            else
            {
                accumulatedTime += frame.Duration;
            }
        }

        // If no frame was found, then try the last frame,
        // just in case timeIntoAnimation somehow exceeds Duration
        if (currentFrame == null)
        {
            currentFrame = frames.LastOrDefault ();
        }

        // If we found a frame, return its rectangle, otherwise
        // return an empty rectangle (one with no width or height)
        if (currentFrame != null)
        {
            return currentFrame.SourceRectangle;
        }
        else
        {
            return Rectangle.Empty;
        }
    }
}
```

Adding the first Animation to CharacterEntity

The `CharacterEntity` will contain animations for walking and standing, as well as a reference to the current `Animation` being displayed.

First, we'll add our first `Animation,` which we'll use to test out the functionality as written so far. Let's add the following members to the CharacterEntity class:

```
Animation walkDown;
Animation currentAnimation;
```

Next, let's define the `walkDown` animation. To do this modify the `CharacterEntity` constructor as follows:

```
public CharacterEntity (GraphicsDevice graphicsDevice)
{
    if (characterSheetTexture == null)
    {
        using (var stream = TitleContainer.OpenStream ("Content/charactersheet.png"))
        {
            characterSheetTexture = Texture2D.FromStream (graphicsDevice, stream);
        }
    }

    walkDown = new Animation ();
    walkDown.AddFrame (new Rectangle (0, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkDown.AddFrame (new Rectangle (16, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkDown.AddFrame (new Rectangle (0, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkDown.AddFrame (new Rectangle (32, 0, 16, 16), TimeSpan.FromSeconds (.25));
}
```

As mentioned earlier, we need to call `Animation.Update` for time-based animations to play. We also need to assign the `currentAnimation`. For now we'll assign the `currentAnimation` to `walkDown`, but we'll be replacing this code later when we implement our movement logic. We'll add the `Update` method to `CharacterEntity` as follows:

```
public void Update(GameTime gameTime)
{
    // temporary - we'll replace this with logic based off of which way the
    // character is moving when we add movement logic
    currentAnimation = walkDown;

    currentAnimation.Update (gameTime);
}
```

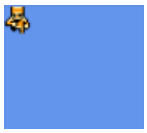
Now that we have the `currentAnimation` being assigned and updated, we can use it to perform our drawing. We'll modify `CharacterEntity.Draw` as follows:

```
public void Draw(SpriteBatch spriteBatch)
{
    Vector2 topLeftOfSprite = new Vector2 (this.X, this.Y);
    Color tintColor = Color.White;
    var sourceRectangle = currentAnimation.CurrentRectangle;

    spriteBatch.Draw(characterSheetTexture, topLeftOfSprite, sourceRectangle, Color.White);
}
```

The last step to getting the `CharacterEntity` animating is to call the newly added `Update` method from `Game1`. Modify `Game1.Update` as follows:

```
protected override void Update(GameTime gameTime)
{
    character.Update (gameTime);
    base.Update(gameTime);
}
```

Now the `CharacterEntity` will play its `walkDown` animation:



Adding Movement to the Character

Next, we'll be adding movement to our character using touch controls. When the user touches the screen, the character will move towards the point where the screen is touched. If no touches are detected, then the character will stand in place.

Defining GetDesiredVelocityFromInput

We'll be using MonoGame's `TouchPanel` class, which provides information about the current state of the touch screen. Let's add a method which will check the `TouchPanel` and return our character's desired velocity:

```
Vector2 GetDesiredVelocityFromInput()
{
    Vector2 desiredVelocity = new Vector2 ();

    TouchCollection touchCollection = TouchPanel.GetState();

    if (touchCollection.Count > 0)
    {
        desiredVelocity.X = touchCollection [0].Position.X - this.X;
        desiredVelocity.Y = touchCollection [0].Position.Y - this.Y;

        if (desiredVelocity.X != 0 || desiredVelocity.Y != 0)
        {
            desiredVelocity.Normalize();
            const float desiredSpeed = 200;
            desiredVelocity *= desiredSpeed;
        }
    }

    return desiredVelocity;
}
```

The `TouchPanel.GetState` method returns a `TouchCollection` which contains information about where the user is touching the screen. If the user is not touching the screen, then the `TouchCollection` will be empty, in which case we shouldn't move the character.

If the user is touching the screen, we will move the character towards the first touch, in other words, the `TouchLocation` at index 0. Initially we'll set the desired velocity to equal the difference between the character's location and the first touch's location:

```
        desiredVelocity.X = touchCollection [0].Position.X - this.X;
        desiredVelocity.Y = touchCollection [0].Position.Y - this.Y;
```

What follows is a bit of math which will keep the character moving at the same speed. To help explain why this is important, let's consider a situation where the user is touching the screen 500 pixels away from where the

character is located. The first line where `desiredVelocity.X` is set would assign a value of 500. However, if the user were touching the screen at a distance of only 100 units from the character, then the `desiredVelocity.X` would be set to 100. The result would be that the character's movement speed would respond to how far away the touch point is from the character. Since we want the character to always move at the same speed, we need to modify the desiredVelocity.

The `if (desiredVelocity.X != 0 || desiredVelocity.Y != 0)` statement is checking if the velocity is non-zero – in other words, it's checking to make sure that the user is not touching the same spot as the character's current position. If not, then we need to set the character's speed to be constant regardless of how far away the touch is. We accomplish this by normalizing the velocity vector which, results in it being a length of 1. A velocity vector of 1 means that the character will move at 1 pixel per second. We'll speed this up by multiplying the value by the desired speed of 200.

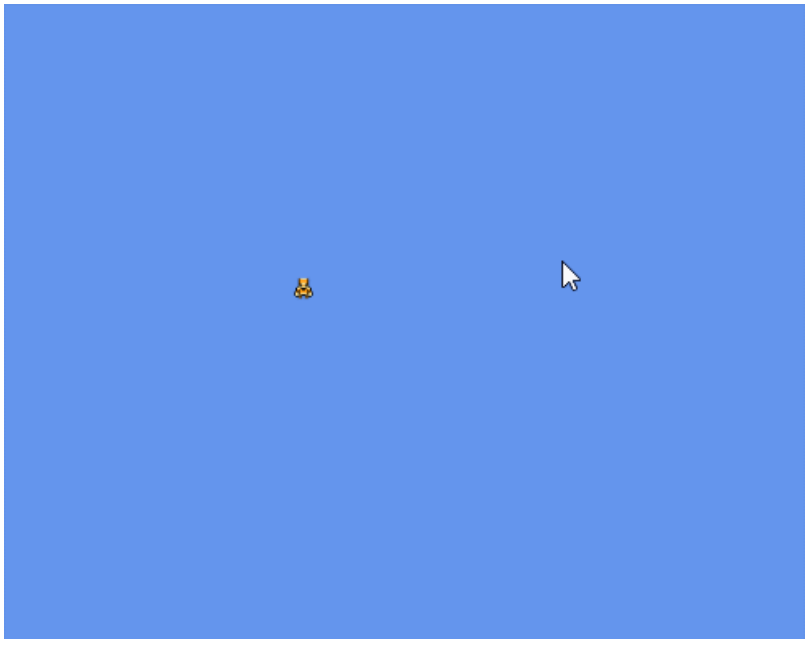
Applying Velocity to Position

The velocity returned from `GetDesiredVelocityFromInput` needs to be applied to the character's `X` and `Y` values to have any effect at runtime. We'll modify the `Update` method as follows:

```
public void Update(GameTime gameTime)
{

    var velocity = GetDesiredVelocityFromInput ();

    this.X += velocity.X * (float)gameTime.ElapsedGameTime.TotalSeconds;
    this.Y += velocity.Y * (float)gameTime.ElapsedGameTime.TotalSeconds;

    // temporary - we'll replace this with logic based off of which way the
    // character is moving when we add movement logic
    currentAnimation = walkDown;

    currentAnimation.Update (gameTime);
}
```

What we've implemented here is called *time-based* movement (as opposed to *frame-based* movement). Time-based movement multiplies a velocity value (in our case the values stored in the `velocity` variable) by how much time has passed since last update which is stored in `gameTime.ElapsedGameTime.TotalSeconds`. If the game runs at fewer frames per second, the elapsed time between frames goes up – the end result is that objects using time-based movement will always move at the same speed regardless of frame rate.

If we run our game now, we'll see that the character is moving towards the touch location:



Matching Movement and Animation

Once we have our character moving and playing a single animation, we can define the remainder of our animations, then use them to reflect the movement of the character. When finished we will have eight animations in total:

- Animations for walking up, down, left, and right
- Animations for standing still and facing up, down, left, and right

Defining the Rest of the Animations

We'll first add the `Animation` instances to the `CharacterEntity` class for all of our animations in the same place where we added `walkDown`. Once we do this, the `CharacterEntity` will have the following `Animation` members:

```
Animation walkDown;
Animation walkUp;
Animation walkLeft;
Animation walkRight;

Animation standDown;
Animation standUp;
Animation standLeft;
Animation standRight;

Animation currentAnimation;
```

Now we'll define the animations in the `CharacterEntity` constructor as follows:

```
public CharacterEntity (GraphicsDevice graphicsDevice)
{
    if (characterSheetTexture == null)
    {
        using (var stream = TitleContainer.OpenStream ("Content/charactersheet.png"))
        {
            characterSheetTexture = Texture2D.FromStream (graphicsDevice, stream);
        }
    }

    walkDown = new Animation ();
    walkDown.AddFrame (new Rectangle (0, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkDown.AddFrame (new Rectangle (16, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkDown.AddFrame (new Rectangle (0, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkDown.AddFrame (new Rectangle (32, 0, 16, 16), TimeSpan.FromSeconds (.25));

    walkUp = new Animation ();
    walkUp.AddFrame (new Rectangle (144, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkUp.AddFrame (new Rectangle (160, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkUp.AddFrame (new Rectangle (144, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkUp.AddFrame (new Rectangle (176, 0, 16, 16), TimeSpan.FromSeconds (.25));

    walkLeft = new Animation ();
    walkLeft.AddFrame (new Rectangle (48, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkLeft.AddFrame (new Rectangle (64, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkLeft.AddFrame (new Rectangle (48, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkLeft.AddFrame (new Rectangle (80, 0, 16, 16), TimeSpan.FromSeconds (.25));

    walkRight = new Animation ();
    walkRight.AddFrame (new Rectangle (96, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkRight.AddFrame (new Rectangle (112, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkRight.AddFrame (new Rectangle (96, 0, 16, 16), TimeSpan.FromSeconds (.25));
    walkRight.AddFrame (new Rectangle (128, 0, 16, 16), TimeSpan.FromSeconds (.25));

    // Standing animations only have a single frame of animation:
    standDown = new Animation ();
    standDown.AddFrame (new Rectangle (0, 0, 16, 16), TimeSpan.FromSeconds (.25));

    standUp = new Animation ();
    standUp.AddFrame (new Rectangle (144, 0, 16, 16), TimeSpan.FromSeconds (.25));

    standLeft = new Animation ();
    standLeft.AddFrame (new Rectangle (48, 0, 16, 16), TimeSpan.FromSeconds (.25));

    standRight = new Animation ();
    standRight.AddFrame (new Rectangle (96, 0, 16, 16), TimeSpan.FromSeconds (.25));
}
```

We should note that the above code was added to the `CharacterEntity` constructor to keep this walkthrough shorter. Games typically will separate the definition of character animations into their own classes or load this information from a data format such as XML or JSON.

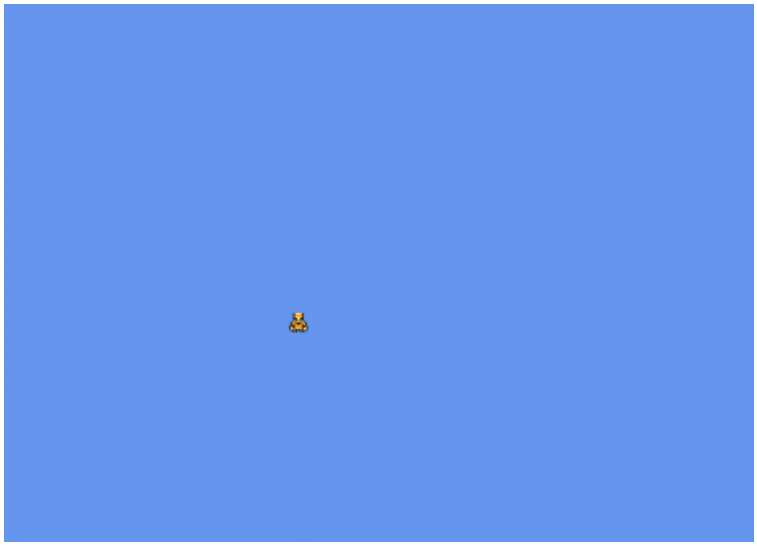
Next, we'll adjust the logic to use the animations according to the direction that the character is moving, or according to the last animation if the character has just stopped. To do this, we'll modify the `Update` method:

```
public void Update(GameTime gameTime)
{
    var velocity = GetDesiredVelocityFromInput ();

    this.X += velocity.X * (float)gameTime.ElapsedGameTime.TotalSeconds;
    this.Y += velocity.Y * (float)gameTime.ElapsedGameTime.TotalSeconds;


    if (velocity != Vector2.Zero)
    {
        bool movingHorizontally = Math.Abs (velocity.X) > Math.Abs (velocity.Y);
        if (movingHorizontally)
        {
            if (velocity.X > 0)
            {
                currentAnimation = walkRight;
            }
            else
            {
                currentAnimation = walkLeft;
            }
        }
        else
        {
            if (velocity.Y > 0)
            {
                currentAnimation = walkDown;
            }
            else
            {
                currentAnimation = walkUp;
            }
        }
    }
    else
    {
        // If the character was walking, we can set the standing animation
        // according to the walking animation that is playing:
        if (currentAnimation == walkRight)
        {
            currentAnimation = standRight;
        }
        else if (currentAnimation == walkLeft)
        {
            currentAnimation = standLeft;
        }
        else if (currentAnimation == walkUp)
        {
            currentAnimation = standUp;
        }
        else if (currentAnimation == walkDown)
        {
            currentAnimation = standDown;
        }
        else if (currentAnimation == null)
        {
            currentAnimation = standDown;
        }

        // if none of the above code hit then the character
        // is already standing, so no need to change the animation.
    }

    currentAnimation.Update (gameTime);
}
```

The code to switch the animations is broken up into two blocks. The first checks if the velocity is not equal to `Vector2.Zero` – this tells us whether the character is moving. If the character is moving, then we can look at the `velocity.X` and `velocity.Y` values to determine which walking animation to play.

If the character is not moving, then we want to set the character's `currentAnimation` to a standing animation – but we only do it if the `currentAnimation` is a walking animation or if an animation hasn't been set. If the `currentAnimation` is not one of the four walking animations, then the character is already standing so we don't need to change `currentAnimation`.

The result of this code is that the character will properly animate when walking, and then face the last direction it was walking when it stops:



Summary

This walkthrough showed how to work with MonoGame to create a cross-platform game with sprites, moving objects, input detection, and animation. It covers creating a general-purpose animation class. It also showed how to create a character entity for organizing code logic.

Related Links

- CharacterSheet Image Resource (sample)
- Walking Game Complete (sample)

Introduction to 3D Graphics with MonoGame

10/3/2018 • 2 minutes to read • Edit Online

MonoGame offers a flexible, efficient API for displaying real-time 3D graphics. It includes higher-level constructs for rendering and also access to lower-level graphics resources.

The MonoGame API provides an extensive set of classes for developing 3D games and applications. It enables direct access to the hardware for maximum performance while maintaining an identical syntax across various platforms.

MonoGame is nearly identical to Microsoft's XNA, so developers experienced with XNA will find MonoGame development familiar. Developers who have not used XNA, but have used DirectX or OpenGL for 3D games, will find many of the classes and concepts familiar as well.

The first section covers how to add a 3D model to your game from an .fbx file. The next section discusses how to create a 3D camera which includes common controls, such as moving and looking around. The last section takes a deeper dive into the `VertexBuffer` class which allows for more control over 3D rendering compared to rendering models loaded from .fbx files.

Topics

Using the Model Class

10/3/2018 • 6 minutes to read • Edit Online

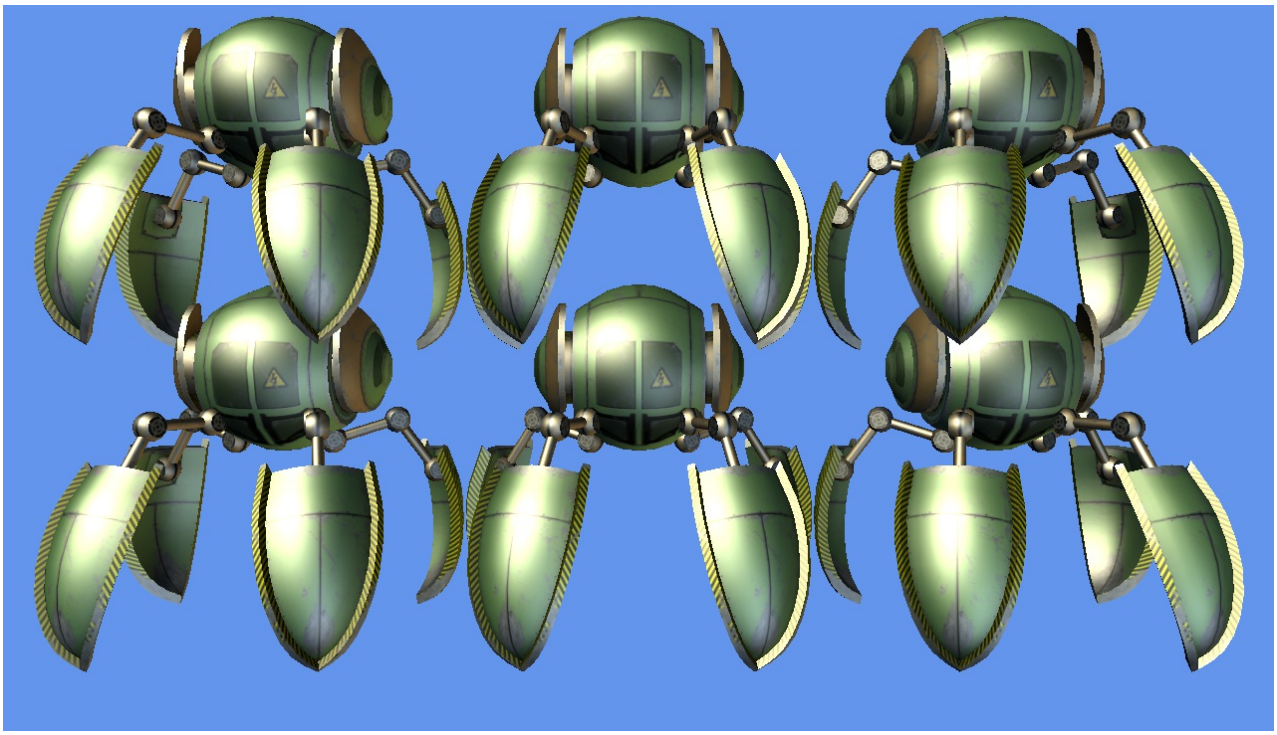
The Model class greatly simplifies rendering complex 3D objects when compared to the traditional method of rendering 3D graphics. Model objects are created from content files, allowing for easy integration of content with no custom code.

The MonoGame API includes a `Model` class which can be used to store data loaded from a content file and to perform rendering. Model files may be very simple (such as a solid colored triangle) or may include information for complex rendering, including texturing and lighting.

This walkthrough uses a 3D model of a robot and covers the following:

- Starting a new game project
- Creating XNBs for the model and its texture
- Including the XNBs in the game project
- Drawing a 3D Model
- Drawing multiple Models

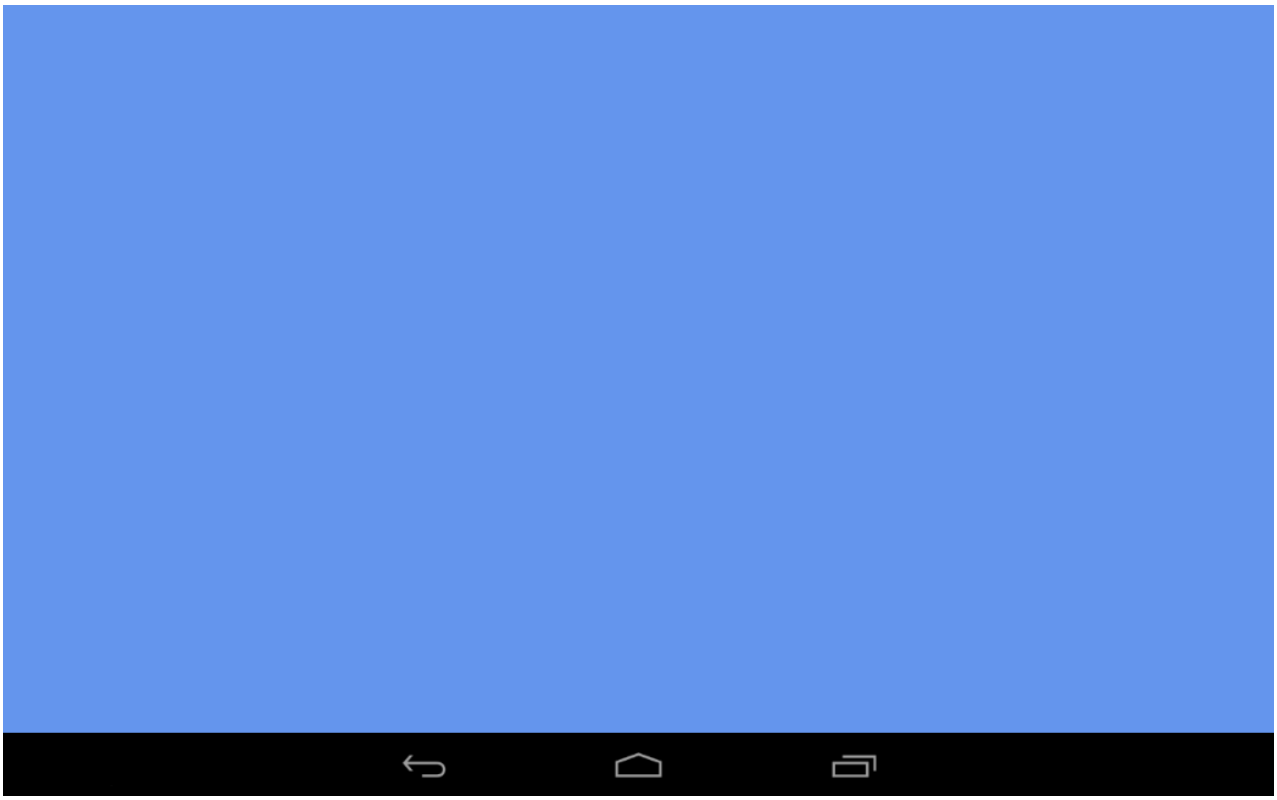
When finished, our project will appear as follows:



Creating an empty game project

We'll need to set up a game project first called MonoGame3D. For information on creating a new MonoGame project, see this walkthrough on creating a Cross Platform Monogame Project.

Before moving on we should verify that the project opens and deploys correctly. Once deployed we should see an empty blue screen:



Including the XNBs in the game project

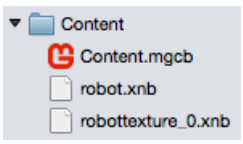
The .xnb file format is a standard extension for built content (content which has been created by the MonoGame Pipeline Tool). All built content has a source file (which is an .fbx file in the case of our model) and a destination file (an .xnb file). The .fbx format is a common 3D model format which can be created by applications such as Maya and Blender.

The `Model` class can be constructed by loading an .xnb file from a disk that contains 3D geometry data. This .xnb file is created through a content project. Monogame templates automatically include a content project (with the extension .mgcp) in our Content folder. For a detailed discussion on the MonoGame Pipeline tool, see the Content Pipeline guide.

For this guide we'll skip over using the MonoGame Pipeline tool and will use the .XNB files included here. Note that the .XNB files differ per platform, so be sure to use the correct set of XNB files for whichever platform you are working with.

We'll unzip the Content.zip file so that we can use the contained .xnb files in our game. If working on an Android project, right-click on the **Assets** folder in the **WalkingGame.Android** project. If working on an iOS project, right-click on the **WalkingGame.iOS** project. Select **Add->Add Files...** and select both .xnb files in the folder for the platform you are working on.

The two files should be part of our project now:



Visual Studio for Mac may not automatically set the build action for newly-added XNBs. For iOS, right-click on each of the files and select **Build Action->BundleResource**. For Android, right-click on each of the files and select **Build Action->AndroidAsset**.

Rendering a 3D model

The last step necessary to see the model on-screen is to add the loading and drawing code. Specifically, we'll be doing the following:

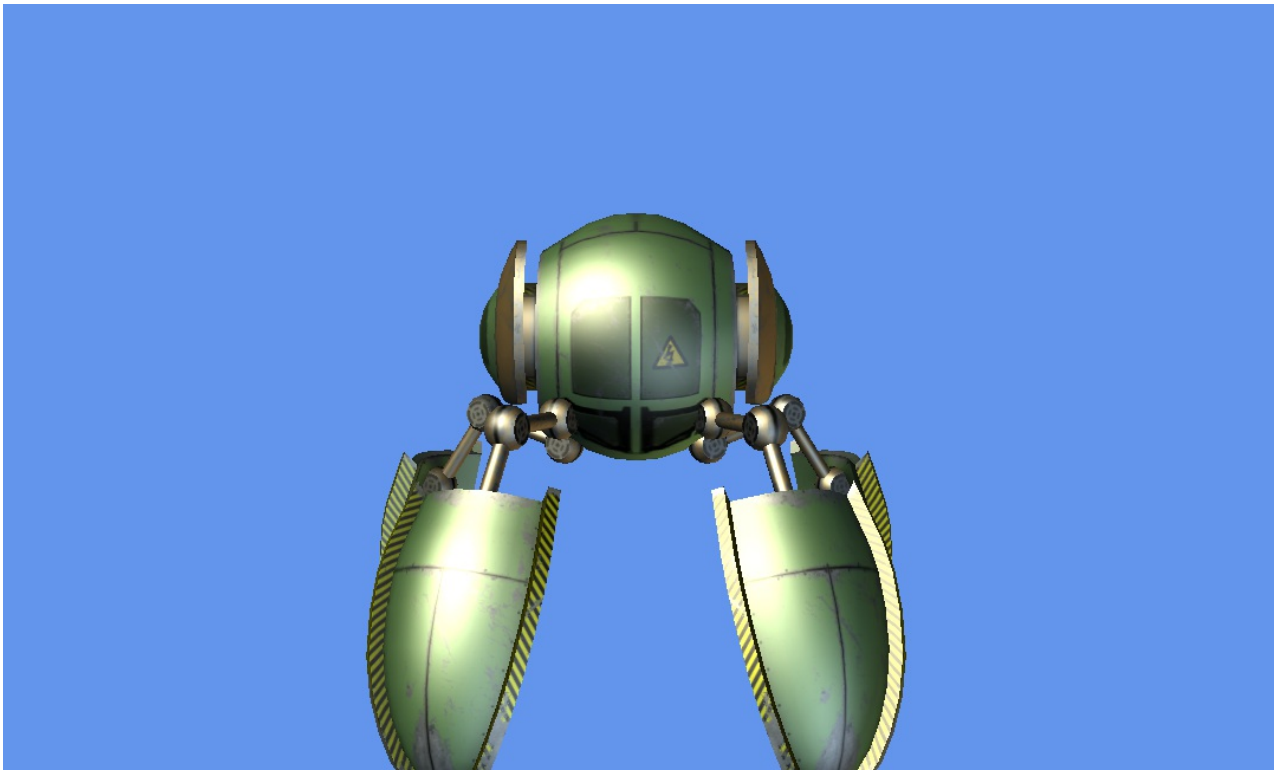
- Defining a `Model` instance in our `Game1` class
- Loading the `Model` instance in `Game1.LoadContent`
- Drawing the `Model` instance in `Game1.Draw`

Replace the `Game1.cs` code file (which is located in the **WalkingGame** PCL) with the following:

```
public class Game1 : Game
{
    GraphicsDeviceManager graphics;

    // This is the model instance that we'll load
    // our XNB into:
    Model model;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        graphics.IsFullScreen = true;

        Content.RootDirectory = "Content";
    }
    protected override void LoadContent()
    {
        // Notice that loading a model is very similar
        // to loading any other XNB (like a Texture2D).
        // The only difference is the generic type.
        model = Content.Load<Model> ("robot");
    }

    protected override void Update(GameTime gameTime)
    {
        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        // A model is composed of "Meshes" which are
        // parts of the model which can be positioned
        // independently, which can use different textures,
        // and which can have different rendering states
        // such as lighting applied.
        foreach (var mesh in model.Meshes)
        {
            // "Effect" refers to a shader. Each mesh may
            // have multiple shaders applied to it for more
            // advanced visuals.
            foreach (BasicEffect effect in mesh.Effects)
            {
                // We could set up custom lights, but this
                // is the quickest way to get somethign on screen:
                effect.EnableDefaultLighting ();
                // This makes lighting look more realistic on
                // round surfaces, but at a slight performance cost:
                effect.PreferPerPixelLighting = true;

                // The world matrix can be used to position, rotate
                // or resize (scale) the model. Identity means that
                // the model is unrotated, drawn at the origin, and
                // its size is unchanged from the loaded content file.
                effect.World = Matrix.Identity;
```

```
                // Move the camera 8 units away from the origin:
                var cameraPosition = new Vector3 (0, 8, 0);
                // Tell the camera to look at the origin:
                var cameraLookAtVector = Vector3.Zero;
                // Tell the camera that positive Z is up
                var cameraUpVector = Vector3.UnitZ;

                effect.View = Matrix.CreateLookAt (
                    cameraPosition, cameraLookAtVector, cameraUpVector);

                // We want the aspect ratio of our display to match
                // the entire screen's aspect ratio:
                float aspectRatio =
                    graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight;
                // Field of view measures how wide of a view our camera has.
                // Increasing this value means it has a wider view, making everything
                // on screen smaller. This is conceptually the same as "zooming out".
                // It also
                float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
                // Anything closer than this will not be drawn (will be clipped)
                float nearClipPlane = 1;
                // Anything further than this will not be drawn (will be clipped)
                float farClipPlane = 200;

                effect.Projection = Matrix.CreatePerspectiveFieldOfView(
                    fieldOfView, aspectRatio, nearClipPlane, farClipPlane);

            }

            // Now that we've assigned our properties on the effects we can
            // draw the entire mesh
            mesh.Draw ();
        }
        base.Draw(gameTime);
    }
}
```

If we run this code we'll see the model on-screen:



Model class

The `Model` class is the core class for performing 3D rendering from content files (such as .fbx files). It contains all

of the information necessary for rendering, including the 3D geometry, the texture references, and `BasicEffect` instances which control positioning, lighting, and camera values.

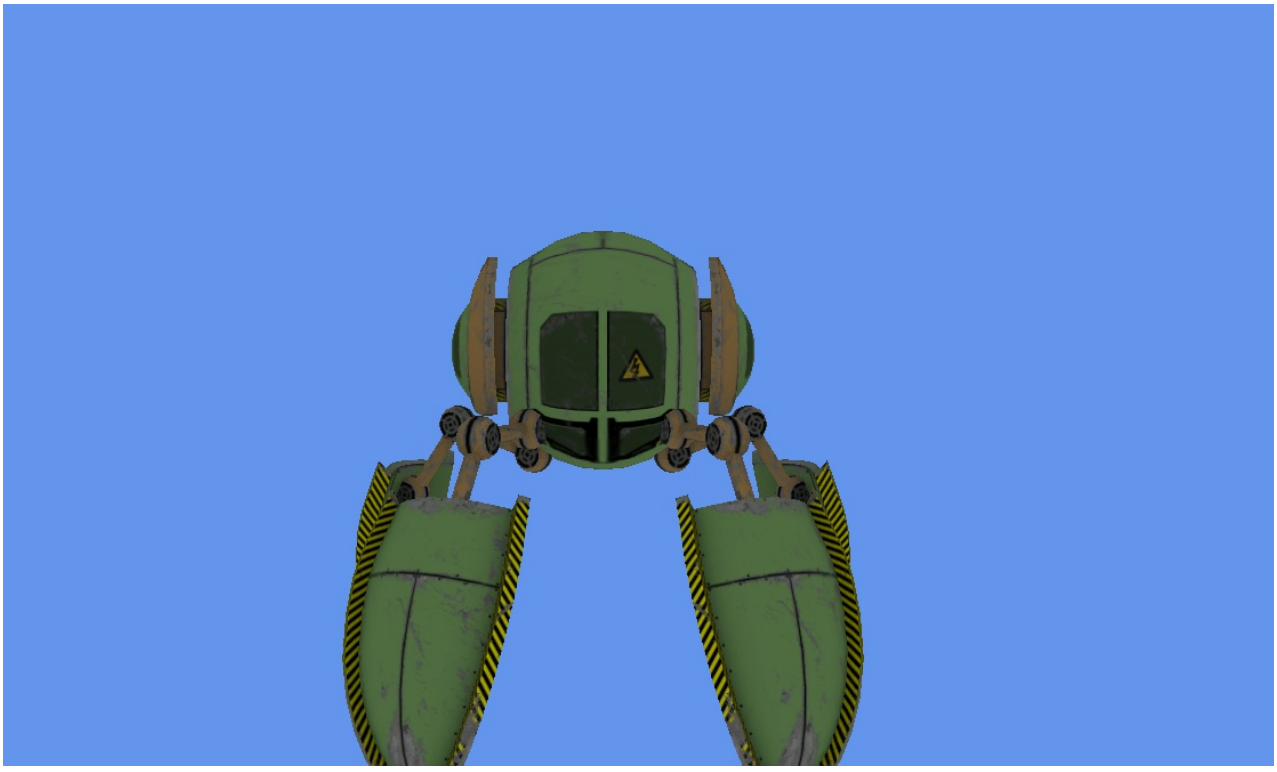
The `Model` class itself does not directly have variables for positioning since a single model instance can be rendered in multiple locations, as we'll show later in this guide.

Each `Model` is composed of one or more `ModelMesh` instances, which are exposed through the `Meshes` property. Although we may consider a `Model` as a single game object (such as a robot or a car), each `ModelMesh` can be drawn with different `BasicEffect` values. For example, individual mesh parts may represent the legs of a robot or the wheels on a car, and we may assign the `BasicEffect` values to make the wheels spin or the legs move.

BasicEffect Class

The `BasicEffect` class provides properties for controlling rendering options. The first modification we make to the `BasicEffect` is to call the `EnableDefaultLighting` method. As the name implies, this enables default lighting, which is very handy for verifying that a `Model` appears in-game as expected. If we comment out the `EnableDefaultLighting` call, then we'll see the model rendered with just its texture, but with no shading or specular glow:

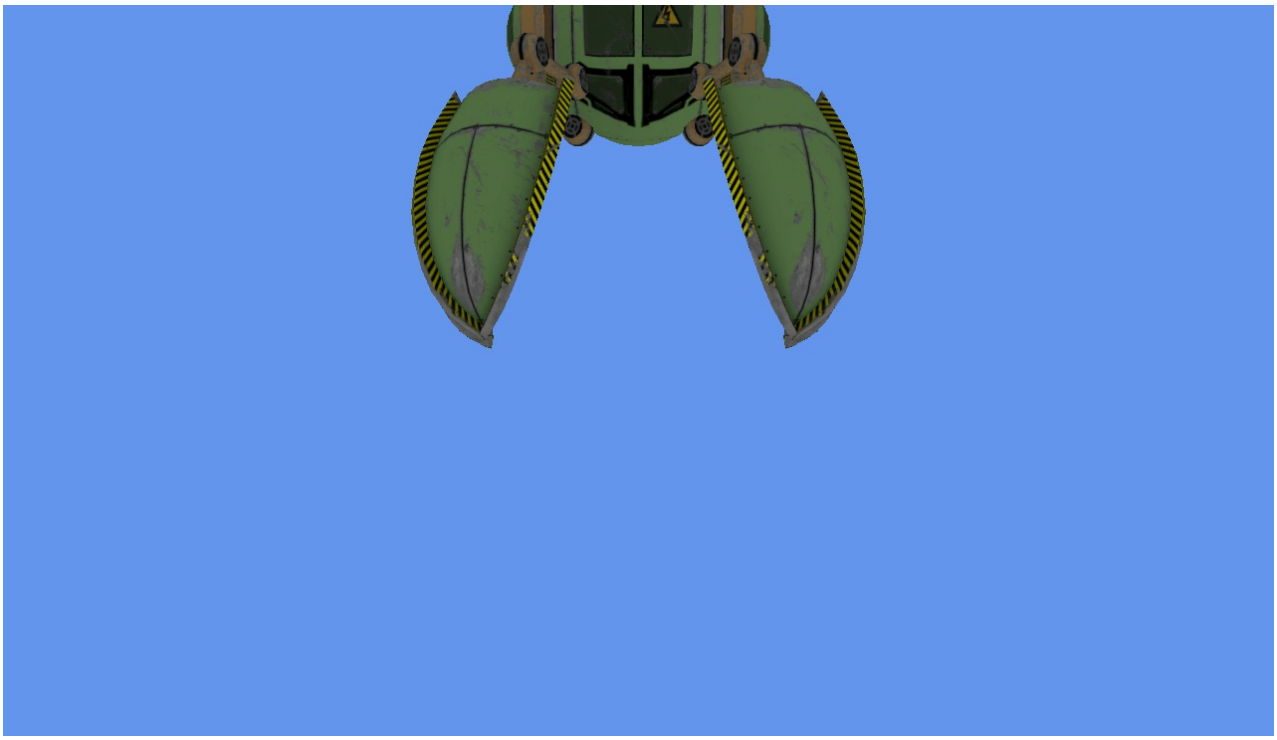
```
//effect.EnableDefaultLighting ();
```



The `World` property can be used to adjust the position, rotation, and scale of the model. The code above uses the `Matrix.Identity` value, which means that the `Model` will render in-game exactly as specified in the .fbx file. We'll be covering matrices and 3D coordinates in more detail in part 3, but as an example we can change the position of the `Model` by changing the `World` property as follows:

```
// Z is up, so changing Z to 3 moves the object up 3 units:
var modelPosition = new Vector3 (0, 0, 3);
effect.World = Matrix.CreateTranslation (modelPosition);
```

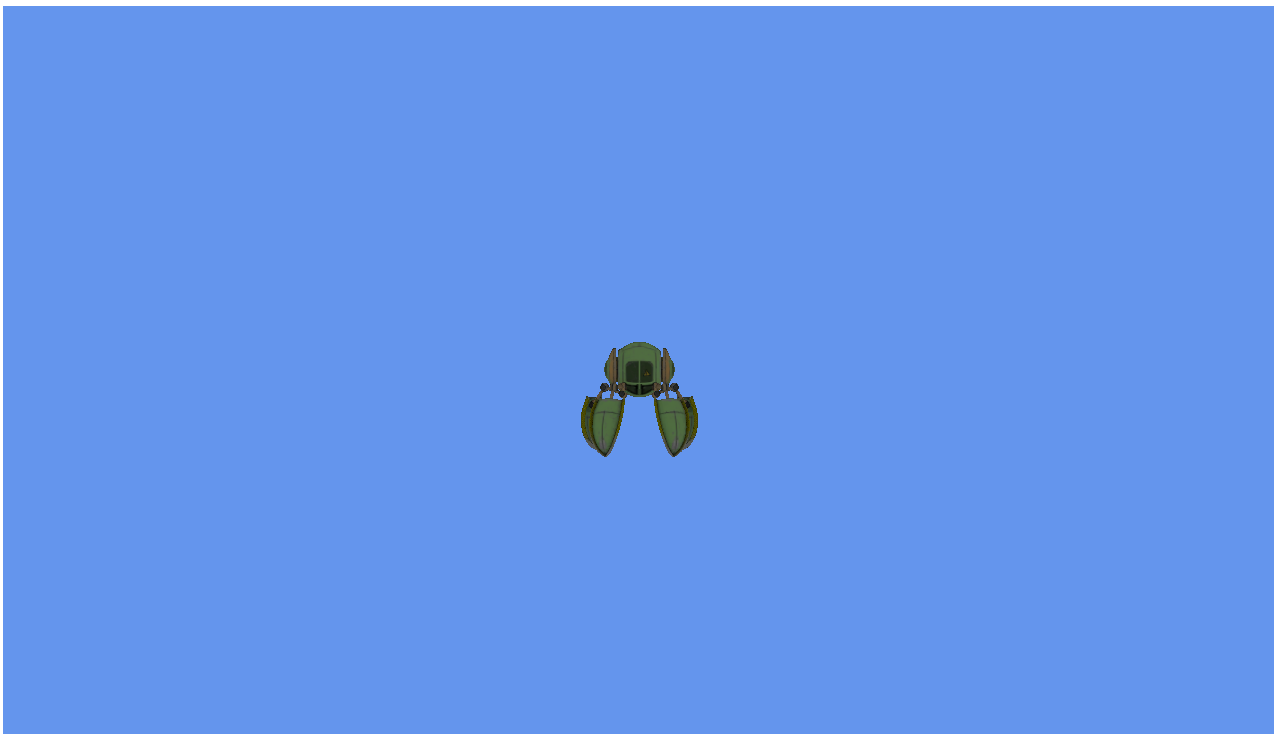
This code results in the object being moved up by 3 world units:



The final two properties assigned on the `BasicEffect` are `View` and `Projection`. We'll be covering 3D cameras in part 3, but as an example, we can modify the position of the camera by changing the local `cameraPosition` variable:

```
// The 8 has been changed to a 30 to move the Camera further back
var cameraPosition = new Vector3 (0, 30, 0);
```

We can see the camera has moved further back, resulting in the `Model` appearing smaller due to perspective:

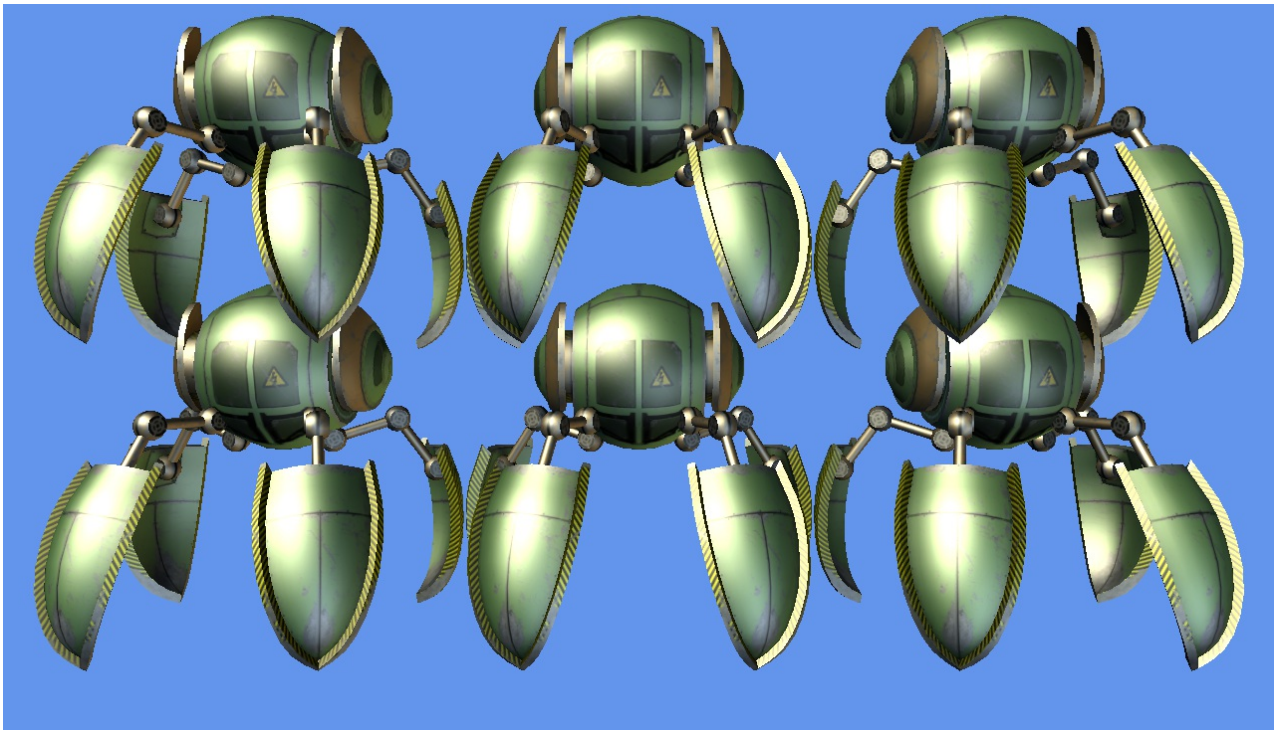


Rendering Multiple Models

As mentioned above, a single `Model` can be drawn multiple times. To make this easier we will be moving the `Model` drawing code into its own method that takes the desired `Model` position as a parameter. Once finished, our `Draw` and `DrawModel` methods will look like:

```
protected override void Draw(GameTime gameTime)
{
 GraphicsDevice.Clear(Color.CornflowerBlue);
 DrawModel (new Vector3 (-4, 0, 0));
 DrawModel (new Vector3 ( 0, 0, 0));
 DrawModel (new Vector3 ( 4, 0, 0));
 DrawModel (new Vector3 (-4, 0, 3));
 DrawModel (new Vector3 ( 0, 0, 3));
 DrawModel (new Vector3 ( 4, 0, 3));
 base.Draw(gameTime);
}
void DrawModel(Vector3 modelPosition)
{
 foreach (var mesh in model.Meshes)
 {
  foreach (BasicEffect effect in mesh.Effects)
  {
   effect.EnableDefaultLighting ();
   effect.PreferPerPixelLighting = true;
   effect.World = Matrix.CreateTranslation (modelPosition);
   var cameraPosition = new Vector3 (0, 10, 0);
   var cameraLookAtVector = Vector3.Zero;
   var cameraUpVector = Vector3.UnitZ;
   effect.View = Matrix.CreateLookAt (
    cameraPosition, cameraLookAtVector, cameraUpVector);
   float aspectRatio =
    graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight;
   float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
   float nearClipPlane = 1;
   float farClipPlane = 200;
   effect.Projection = Matrix.CreatePerspectiveFieldOfView(
    fieldOfView, aspectRatio, nearClipPlane, farClipPlane);
  }
  // Now that we've assigned our properties on the effects we can
  // draw the entire mesh
  mesh.Draw ();
 }
}
```

This results in the robot Model being drawn six times:



Summary

This walkthrough introduced MonoGame's `Model` class. It covers converting an .fbx file into an .xnb, which can in-turn be loaded into a `Model` class. It also shows how modifications to a `BasicEffect` instance can impact `Model` drawing.

Related Links

- MonoGame Model Reference
- Content.zip
- Completed project (sample)

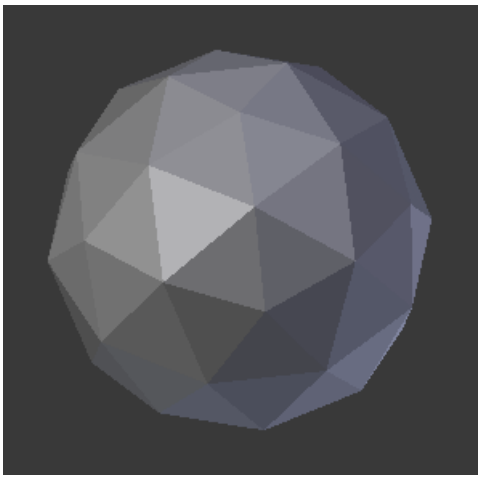
Drawing 3D Graphics with Vertices in MonoGame

10/3/2018 • 9 minutes to read • Edit Online

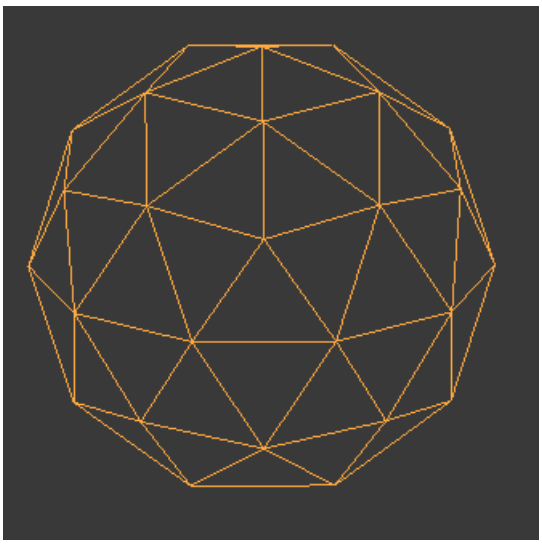
MonoGame supports using arrays of vertices to define how a 3D object is rendered on a per-point basis. Users can take advantage of vertex arrays to create dynamic geometry, implement special effects, and improve the efficiency of their rendering through culling.

Users who have read through the guide on rendering Models will be familiar with rendering a 3D model in MonoGame. The `Model` class is an effective way to render 3D graphics when working with data defined in a file (such as .fbx), and when dealing with static data. Some games require 3D geometry to be defined or manipulated dynamically at runtime. In these cases, we can use arrays of *vertices* to define and render geometry. A vertex is a general term for a point in 3D space which is part of an ordered list used to define geometry. Typically vertices are ordered in such a way as to define a series of triangles.

To help visualize how vertices are used to create 3D objects, let's consider the following sphere:



As shown above, the sphere is clearly composed of multiple triangles. We can view the wireframe of the sphere to see how the vertices connect to form triangles:

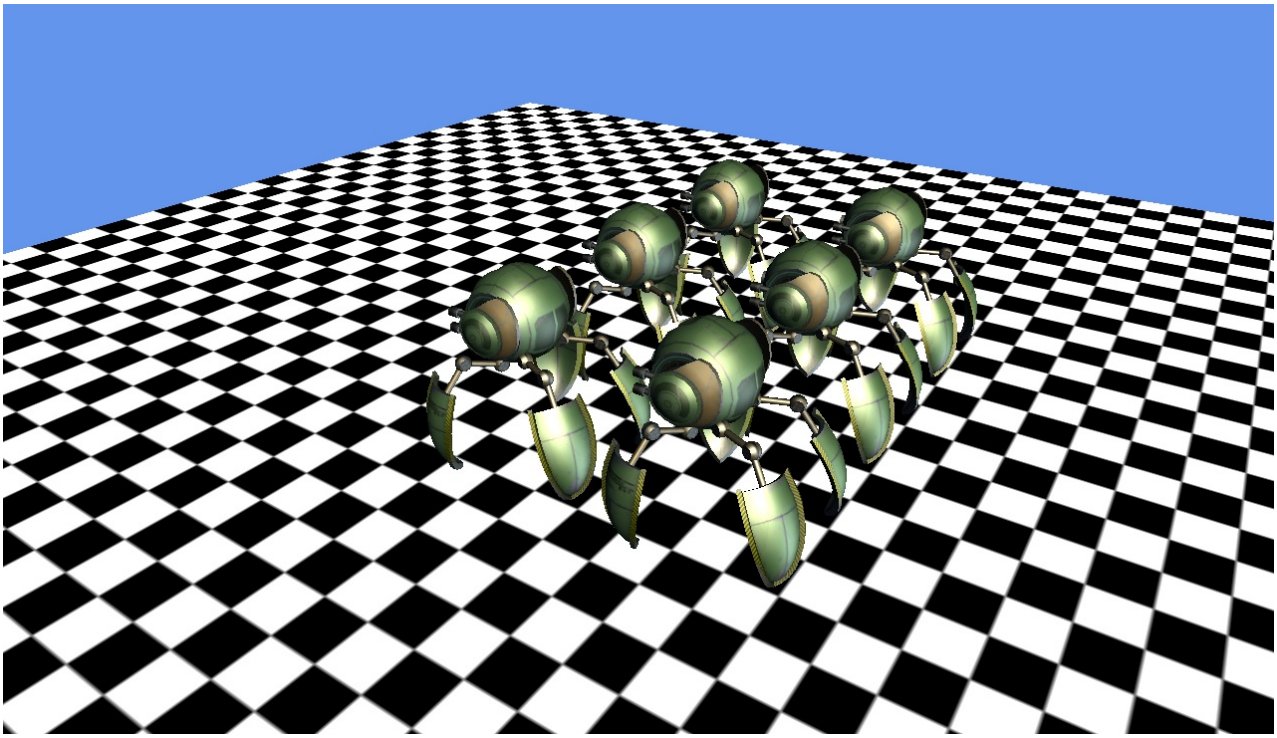


This walkthrough will cover the following topics:

- Creating a project
- Creating the vertices
- Adding drawing code

- Rendering with a texture
- Modifying texture coordinates
- Rendering vertices with models

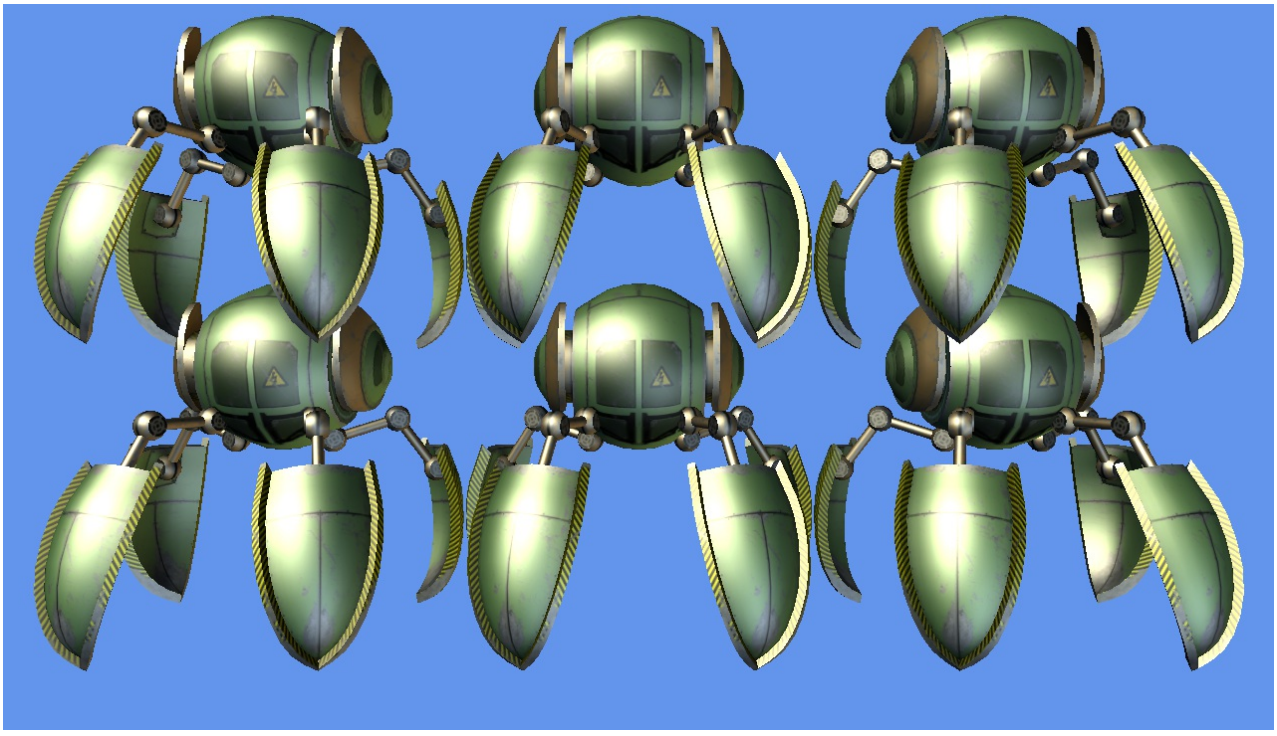
The finished project will contain a checkered floor which will be drawn using a vertex array:



Creating a Project

First, we'll download a project which will serve as our starting point. We'll use the Model project which can be found here.

Once downloaded and unzipped, open and run the project. We expect to see six robot models being drawn on-screen:



By the end of this project we'll be combining our own custom vertex rendering with the robot `Model`, so we aren't

going to delete the robot rendering code. Instead, we'll just clear out the `Game1.Draw` call to remove the drawing of the 6 robots for now. To do this, open the **Game1.cs** file and locate the `Draw` method. Modify it so it contains the following code:

```
protected override void Draw(GameTime gameTime)
{
  GraphicsDevice.Clear(Color.CornflowerBlue);
  base.Draw(gameTime);
}
```

This will result in our game displaying an empty blue screen:

Creating the Vertices

We will create an array of vertices to define our geometry. In this walkthrough, we'll be creating a 3D plane (a square in 3D space, not an airplane). Although our plane has four sides and four corners, it will be composed of two triangles, each of which requires three vertices. Therefore, we will be defining a total of six points.

So far we've been talking about vertices in a general sense, but MonoGame provides some standard structs which can be used for vertices:

- `Microsoft.Xna.Framework.Graphics.VertexPositionColor`
- `Microsoft.Xna.Framework.Graphics.VertexPositionColorTexture`
- `Microsoft.Xna.Framework.Graphics.VertexPositionNormalTexture`
- `Microsoft.Xna.Framework.Graphics.VertexPositionTexture`

Each type's name indicates the components it contains. For example, `VertexPositionColor` contains values for position and color. Let's look at each of the components:

- Position – All vertex types include a `Position` component. The `Position` values define where the vertex is located in 3D space (X, Y, and Z).
- Color – Vertices can optionally specify a `Color` value to perform custom tinting.

- Normal – Normals define which way the surface of the object is facing. Normals are necessary if rendering an object with lighting since the direction that a surface is facing impacts how much light it receives. Normals are typically specified as a *unit vector* – a 3D vector which has a length of 1.
- Texture – Texture refers to texture coordinates – that is, which portion of a texture should appear at a given vertex. Texture values are necessary if rendering a 3D object with a texture. Texture coordinates are normalized coordinates, which means that values will fall between 0 and 1. We'll cover texture coordinates in more detail later in this guide.

Our plane will serve as a floor, and we'll want to apply a texture when performing our rendering, so we will use the `VertexPositionTexture` type to define our vertices.

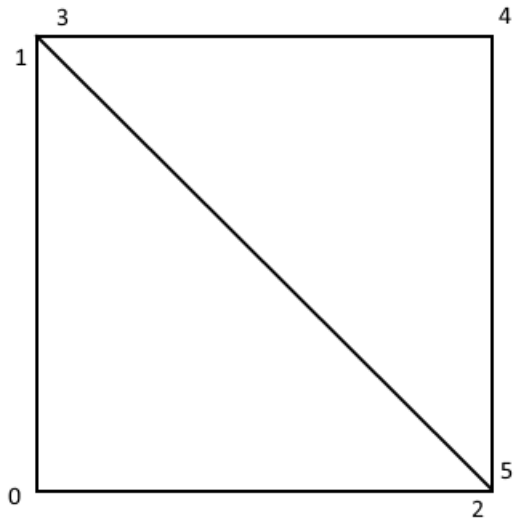
First, we'll add a member to our Game1 class:

```
VertexPositionTexture[] floorVerts;
```

Next, define our vertices in `Game1.Initialize`. Notice that the provided template referenced earlier in this article does not contain a `Game1.Initialize` method, so we need to add the entire method to `Game1`:

```
protected override void Initialize ()
{
 floorVerts = new VertexPositionTexture[6];
 floorVerts [0].Position = new Vector3 (-20, -20, 0);
 floorVerts [1].Position = new Vector3 (-20,  20, 0);
 floorVerts [2].Position = new Vector3 ( 20, -20, 0);
 floorVerts [3].Position = floorVerts[1].Position;
 floorVerts [4].Position = new Vector3 ( 20,  20, 0);
 floorVerts [5].Position = floorVerts[2].Position;
 // We'll be assigning texture values later
 base.Initialize ();
}
```

To help visualize what our vertices will look like, consider the following diagram:



We need to rely on our diagram to visualize the vertices until we finish implementing our rendering code.

Adding Drawing Code

Now that we have the positions for our geometry defined, we can write our rendering code.

First, we'll need to define a `BasicEffect` instance which will hold parameters for rendering such as position and lighting. To do this, add a `BasicEffect` member to the `Game1` class below where the `floorVerts` field is defined:

```
...
VertexPositionTexture[] floorVerts;
// new code:
BasicEffect effect;
```

Next, modify the `Initialize` method to define the effect:

```
protected override void Initialize ()
{
    floorVerts = new VertexPositionTexture[6];

    floorVerts [0].Position = new Vector3 (-20, -20, 0);
    floorVerts [1].Position = new Vector3 (-20,  20, 0);
    floorVerts [2].Position = new Vector3 ( 20, -20, 0);

    floorVerts [3].Position = floorVerts[1].Position;
    floorVerts [4].Position = new Vector3 ( 20,  20, 0);
    floorVerts [5].Position = floorVerts[2].Position;
    // new code:
    effect = new BasicEffect (graphics.GraphicsDevice);

    base.Initialize ();
}
```

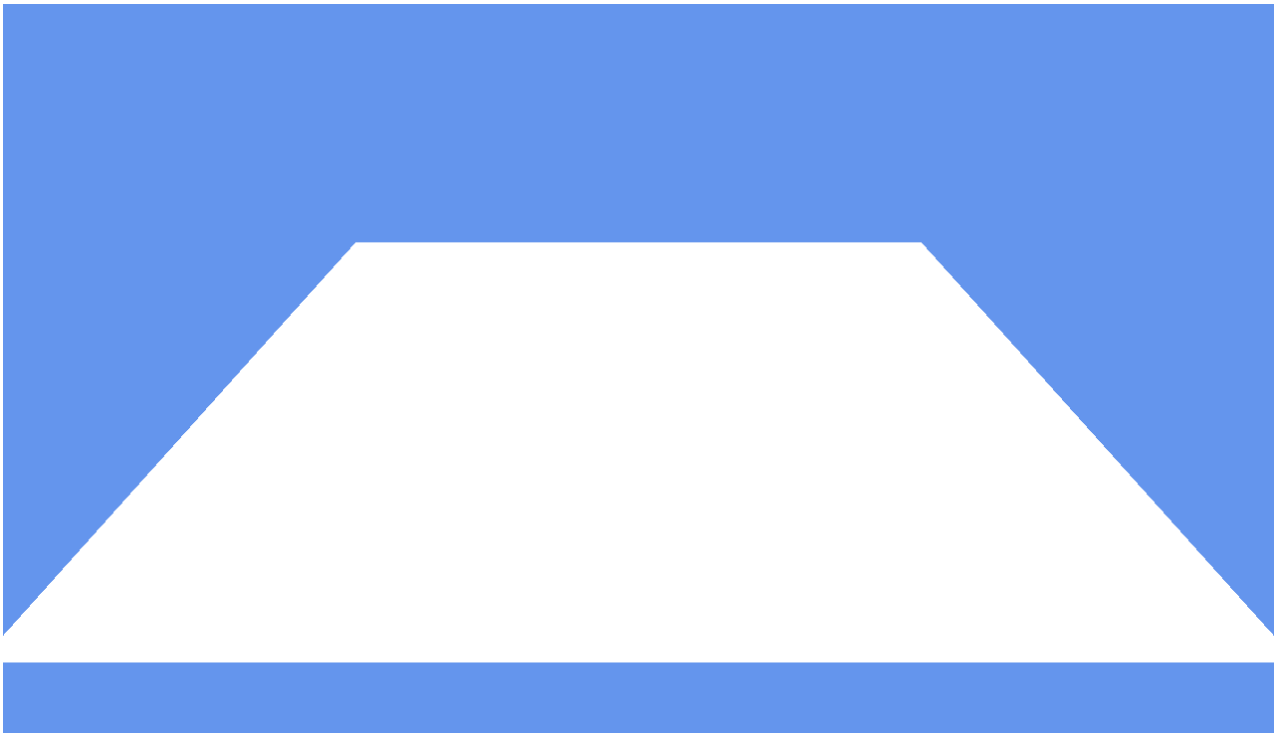
Now we can add code to perform the drawing:

```
void DrawGround()
{
    // The assignment of effect.View and effect.Projection
    // are nearly identical to the code in the Model drawing code.
    var cameraPosition = new Vector3 (0, 40, 20);
    var cameraLookAtVector = Vector3.Zero;
    var cameraUpVector = Vector3.UnitZ;

    effect.View = Matrix.CreateLookAt (
        cameraPosition, cameraLookAtVector, cameraUpVector);

    float aspectRatio =
        graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight;
    float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
    float nearClipPlane = 1;
    float farClipPlane = 200;

    effect.Projection = Matrix.CreatePerspectiveFieldOfView(
        fieldOfView, aspectRatio, nearClipPlane, farClipPlane);


    foreach (var pass in effect.CurrentTechnique.Passes)
    {
        pass.Apply ();

        graphics.GraphicsDevice.DrawUserPrimitives (
            // We'll be rendering two trinalges
            PrimitiveType.TriangleList,
            // The array of verts that we want to render
            floorVerts,
            // The offset, which is 0 since we want to start
            // at the beginning of the floorVerts array
            0,
            // The number of triangles to draw
            2);
    }
}
```

We'll need to call `DrawGround` in our `Game1.Draw`:

```
protected override void Draw (GameTime gameTime)
{
 GraphicsDevice.Clear (Color.CornflowerBlue);

 DrawGround ();

 base.Draw (gameTime);
}
```

The app will display the following when executed:



Let's look at some of the details in the code above.

View and Projection Properties

The `View` and `Projection` properties control how we view the scene. We'll be modifying this code later when we re-add the model rendering code. Specifically, `View` controls the location and orientation of the camera, and `Projection` controls the *field of view* (which can be used to zoom the camera).

Techniques and Passes

Once we've assigned properties on our effect we can perform the actual rendering.

We won't be changing the `CurrentTechnique` property in this walkthrough, but more advanced games may have a single effect which can perform drawing in different ways (such as how the color value is applied). Each of these rendering modes can be represented as a technique which can be assigned prior to rendering. Furthermore, each technique may require multiple passes to render properly. Effects may need multiple passes if rendering complex visuals such as a glowing surface or fur.

The important thing to keep in mind is that the `foreach` loop enables the same C# code to render any effect regardless of the complexity of the underlying `BasicEffect`.

DrawUserPrimitives

`DrawUserPrimitives` is where the vertices are rendered. The first parameter tells the method how we have organized our vertices. We have structured them so that each triangle is defined by three ordered vertices, so we use the `PrimitiveType.TriangleList` value.

The second parameter is the array of vertices which we defined earlier.

The third parameter specifies the first index to draw. Since we want our entire vertex array to be rendered, we'll pass a value of 0.

Finally, we specify how many triangles to render. Our vertex array contains two triangles, so pass a value of 2.

Rendering with a Texture

At this point our app renders a white plane (in perspective). Next we'll add a texture to our project to be used when rendering our plane.

To keep things simple we'll add the .png directly to our project rather than using the MonoGame Pipeline tool. To do this, download this .png file to your computer. Once downloaded, right-click on the **Content** folder in the Solution pad and select **Add>Add Files...** . If working on Android, then this folder will be located under the **Assets** folder in the Android-specific project. If on iOS, then this folder will be in the root of the iOS project. Navigate to the location where **checkerboard.png** is saved and select this file. Select to copy the file to the directory.

Next, we'll add the code to create our `Texture2D` instance. First, add the `Texture2D` as a member of `Game1` under the `BasicEffect` instance:

```
...
BasicEffect effect;
// new code:
Texture2D checkerboardTexture;
```

Modify `Game1.LoadContent` as follows:

```
protected override void LoadContent()
{
    // Notice that loading a model is very similar
    // to loading any other XNB (like a Texture2D).
    // The only difference is the generic type.
    model = Content.Load<Model> ("robot");

    // We aren't using the content pipeline, so we need
    // to access the stream directly:
    using (var stream = TitleContainer.OpenStream ("Content/checkerboard.png"))
    {
        checkerboardTexture = Texture2D.FromStream (this.GraphicsDevice, stream);
    }
}
```

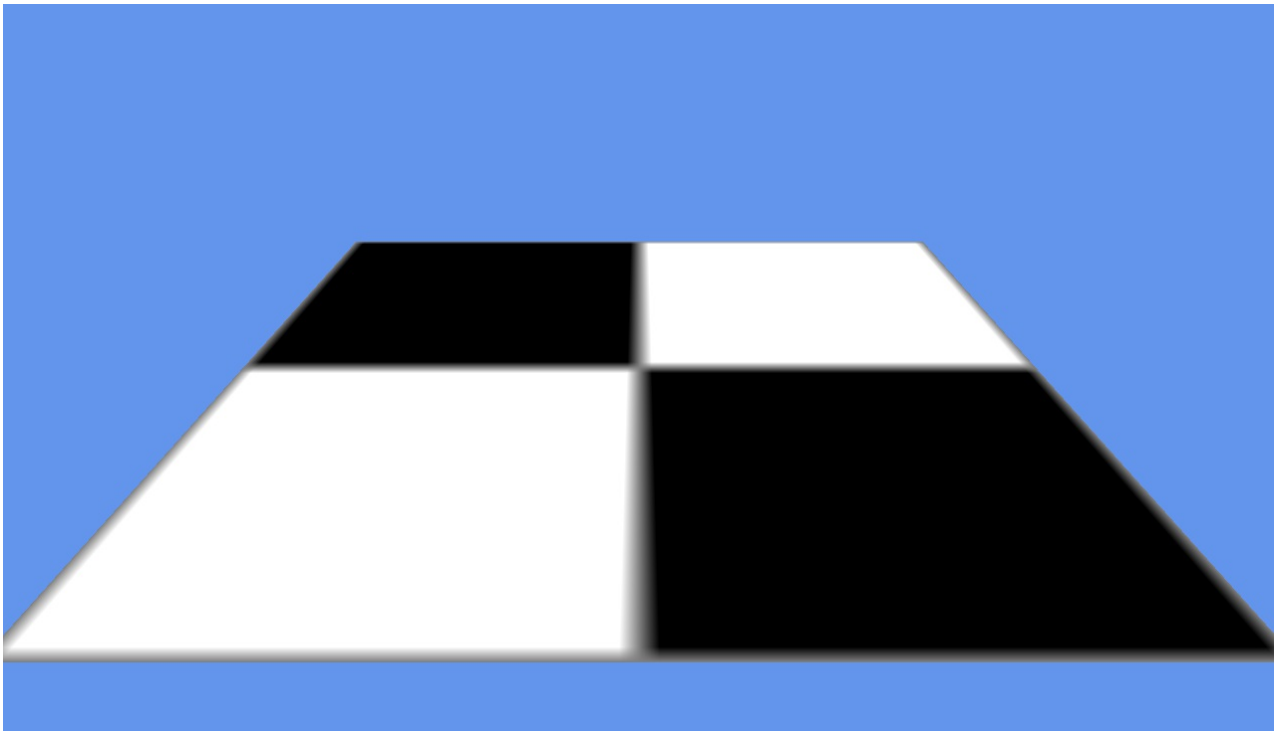
Next, modify the `DrawGround` method. The only modification necessary is to assign `effect.TextureEnabled` to `true` and to set the `effect.Texture` to `checkerboardTexture`:

```
void DrawGround()
{
    // The assignment of effect.View and effect.Projection
    // are nearly identical to the code in the Model drawing code.
    var cameraPosition = new Vector3 (0, 40, 20);
    var cameraLookAtVector = Vector3.Zero;
    var cameraUpVector = Vector3.UnitZ;

    effect.View = Matrix.CreateLookAt (
        cameraPosition, cameraLookAtVector, cameraUpVector);

    float aspectRatio =
        graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight;
    float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
    float nearClipPlane = 1;
    float farClipPlane = 200;

    effect.Projection = Matrix.CreatePerspectiveFieldOfView(
        fieldOfView, aspectRatio, nearClipPlane, farClipPlane);

    // new code:
    effect.TextureEnabled = true;
    effect.Texture = checkerboardTexture;

    foreach (var pass in effect.CurrentTechnique.Passes)
    {
        pass.Apply ();

        graphics.GraphicsDevice.DrawUserPrimitives (
                    PrimitiveType.TriangleList,
            floorVerts,
            0,
            2);
    }
}
```

Finally, we need to modify the `Game1.Initialize` method to also assign texture coordinates on our vertices:

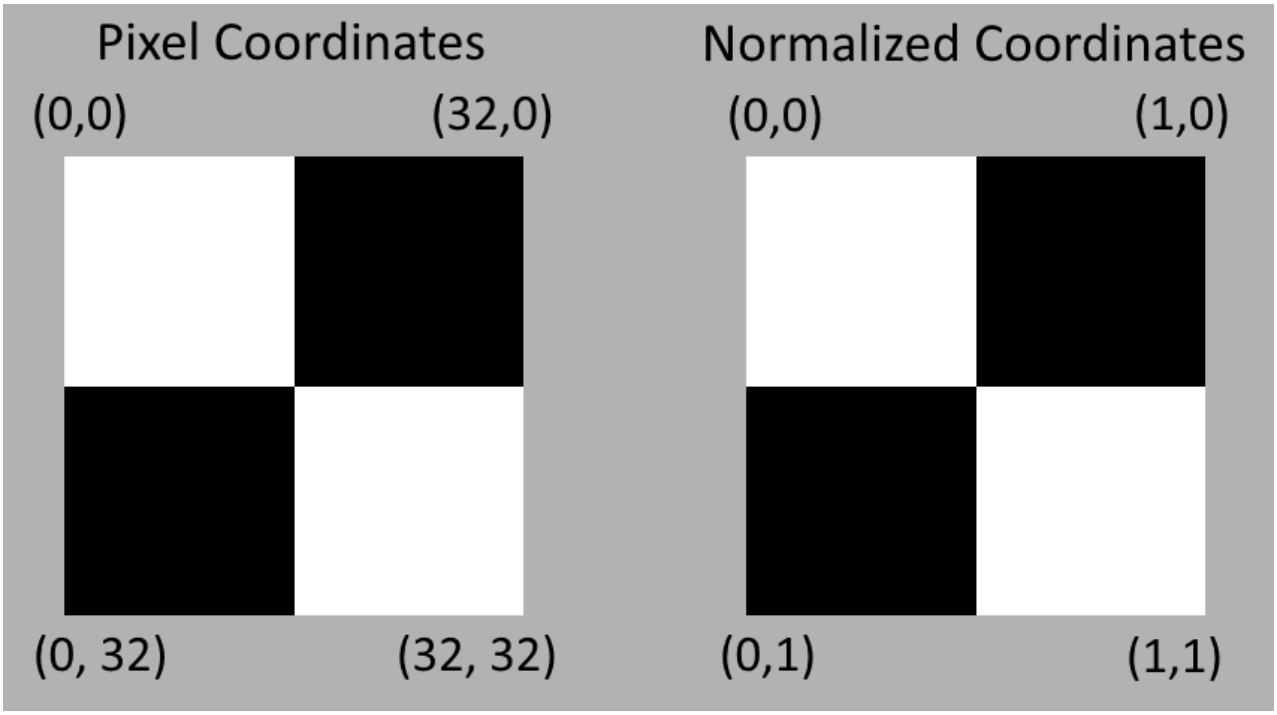
```
protected override void Initialize ()
{
    floorVerts = new VertexPositionTexture[6];

    floorVerts [0].Position = new Vector3 (-20, -20, 0);
    floorVerts [1].Position = new Vector3 (-20,  20, 0);
    floorVerts [2].Position = new Vector3 ( 20, -20, 0);

    floorVerts [3].Position = floorVerts[1].Position;
    floorVerts [4].Position = new Vector3 ( 20,  20, 0);
    floorVerts [5].Position = floorVerts[2].Position;

    // New code:
    floorVerts [0].TextureCoordinate = new Vector2 (0, 0);
    floorVerts [1].TextureCoordinate = new Vector2 (0, 1);
    floorVerts [2].TextureCoordinate = new Vector2 (1, 0);

    floorVerts [3].TextureCoordinate = floorVerts[1].TextureCoordinate;
    floorVerts [4].TextureCoordinate = new Vector2 (1, 1);
    floorVerts [5].TextureCoordinate = floorVerts[2].TextureCoordinate;

    effect = new BasicEffect (graphics.GraphicsDevice);

    base.Initialize ();
}
```

If we run the code, we can see that our plane now displays a checkerboard pattern:



Modifying Texture Coordinates

MonoGame uses normalized texture coordinates, which are coordinates between 0 and 1 rather than between 0 and the texture's width or height. The following diagram can help visualize normalized coordinates:

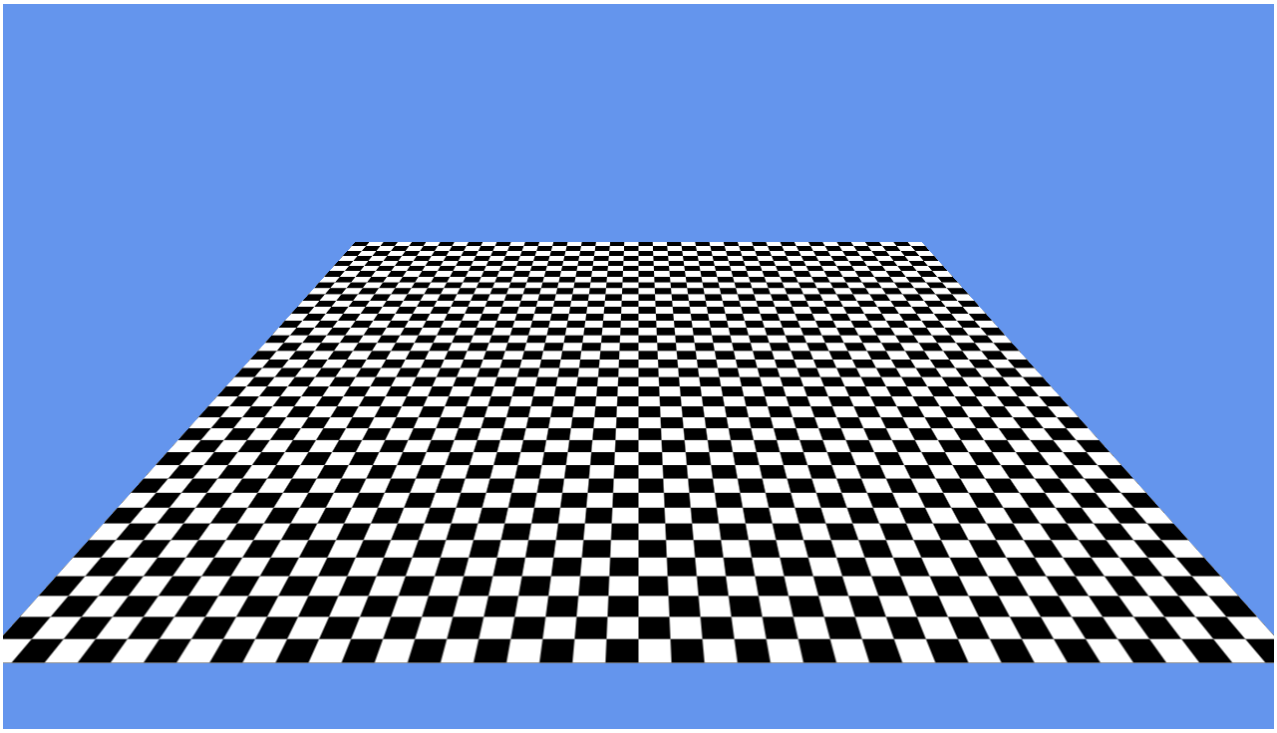


Normalized texture coordinates allow texture resizing without needing to rewrite code or recreate models (such as .fbx files). This is possible because normalized coordinates represent a ratio rather than specific pixels. For example, (1,1) will always represent the bottom-right corner regardless of the texture size.

We can change the texture coordinate assignment to use a single variable for the number of repetitions:

```
protected override void Initialize ()
{
    floorVerts = new VertexPositionTexture[6];

    floorVerts [0].Position = new Vector3 (-20, -20, 0);
    floorVerts [1].Position = new Vector3 (-20,  20, 0);
    floorVerts [2].Position = new Vector3 ( 20, -20, 0);

    floorVerts [3].Position = floorVerts[1].Position;
    floorVerts [4].Position = new Vector3 ( 20,  20, 0);
    floorVerts [5].Position = floorVerts[2].Position;

    int repetitions = 20;

    floorVerts [0].TextureCoordinate = new Vector2 (0, 0);
    floorVerts [1].TextureCoordinate = new Vector2 (0, repetitions);
    floorVerts [2].TextureCoordinate = new Vector2 (repetitions, 0);

    floorVerts [3].TextureCoordinate = floorVerts[1].TextureCoordinate;
    floorVerts [4].TextureCoordinate = new Vector2 (repetitions, repetitions);
    floorVerts [5].TextureCoordinate = floorVerts[2].TextureCoordinate;

    effect = new BasicEffect (graphics.GraphicsDevice);

    base.Initialize ();
}
```

This results in the texture repeating 20 times:



Rendering Vertices with Models

Now that our plane is rendering properly, we can re-add the models to view everything together. First, we'll re-add the model code to our `Game1.Draw` method (with modified positions):

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    DrawGround ();

    DrawModel (new Vector3 (-4, 0, 3));
    DrawModel (new Vector3 ( 0, 0, 3));
    DrawModel (new Vector3 ( 4, 0, 3));

    DrawModel (new Vector3 (-4, 4, 3));
    DrawModel (new Vector3 ( 0, 4, 3));
    DrawModel (new Vector3 ( 4, 4, 3));

    base.Draw(gameTime);
}
```

We will also create a `Vector3` in `Game1` to represent our camera's position. We'll add a field under our `checkerboardTexture` declaration:

```
...
Texture2D checkerboardTexture;
// new code:
Vector3 cameraPosition = new Vector3(0, 10, 10);
```

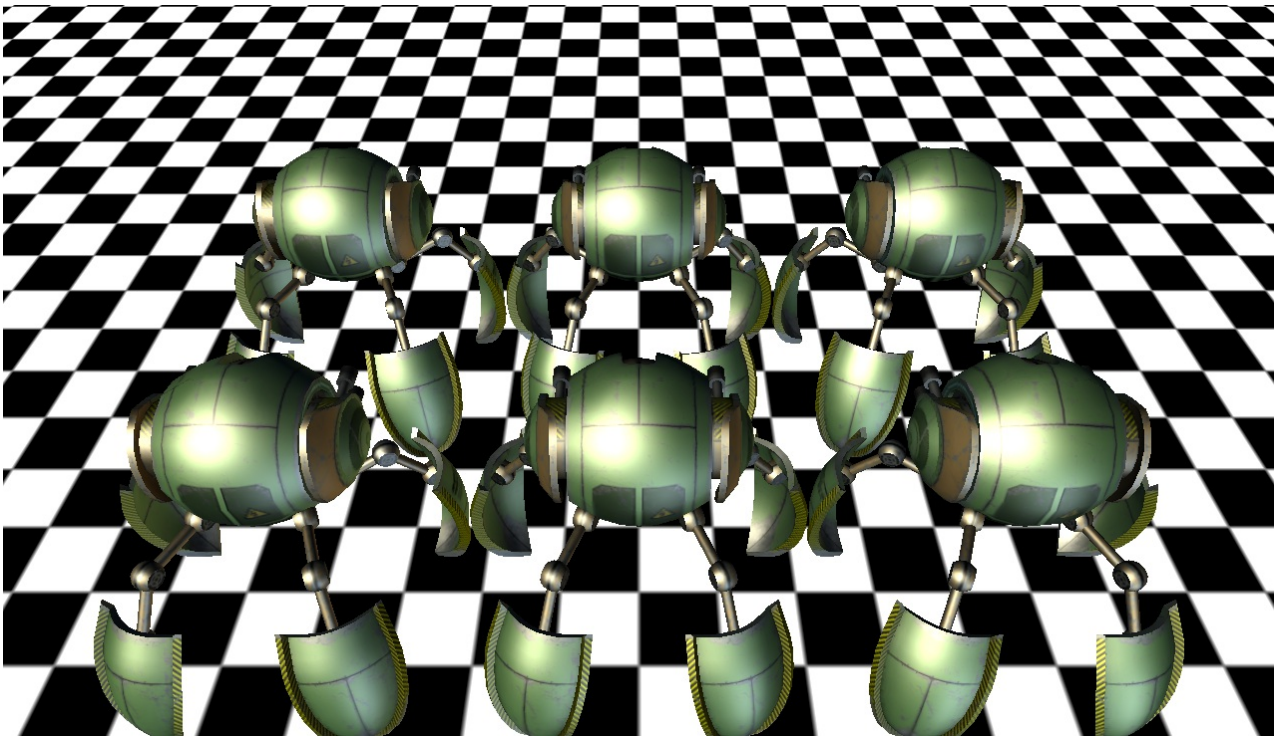
Next, remove the local `cameraPosition` variable from the `DrawModel` method:

```
void DrawModel(Vector3 modelPosition)
{
    foreach (var mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting ();
            effect.PreferPerPixelLighting = true;

            effect.World = Matrix.CreateTranslation (modelPosition);

            var cameraLookAtVector = Vector3.Zero;
            var cameraUpVector = Vector3.UnitZ;

            effect.View = Matrix.CreateLookAt (
                cameraPosition, cameraLookAtVector, cameraUpVector);
            ...
```

Similarly remove the local `cameraPosition` variable from the `DrawGround` method:

```
void DrawGround()
{
    // The assignment of effect.View and effect.Projection
    // are nearly identical to the code in the Model drawing code.
    var cameraLookAtVector = Vector3.Zero;
    var cameraUpVector = Vector3.UnitZ;

    effect.View = Matrix.CreateLookAt (
        cameraPosition, cameraLookAtVector, cameraUpVector);
    ...
```

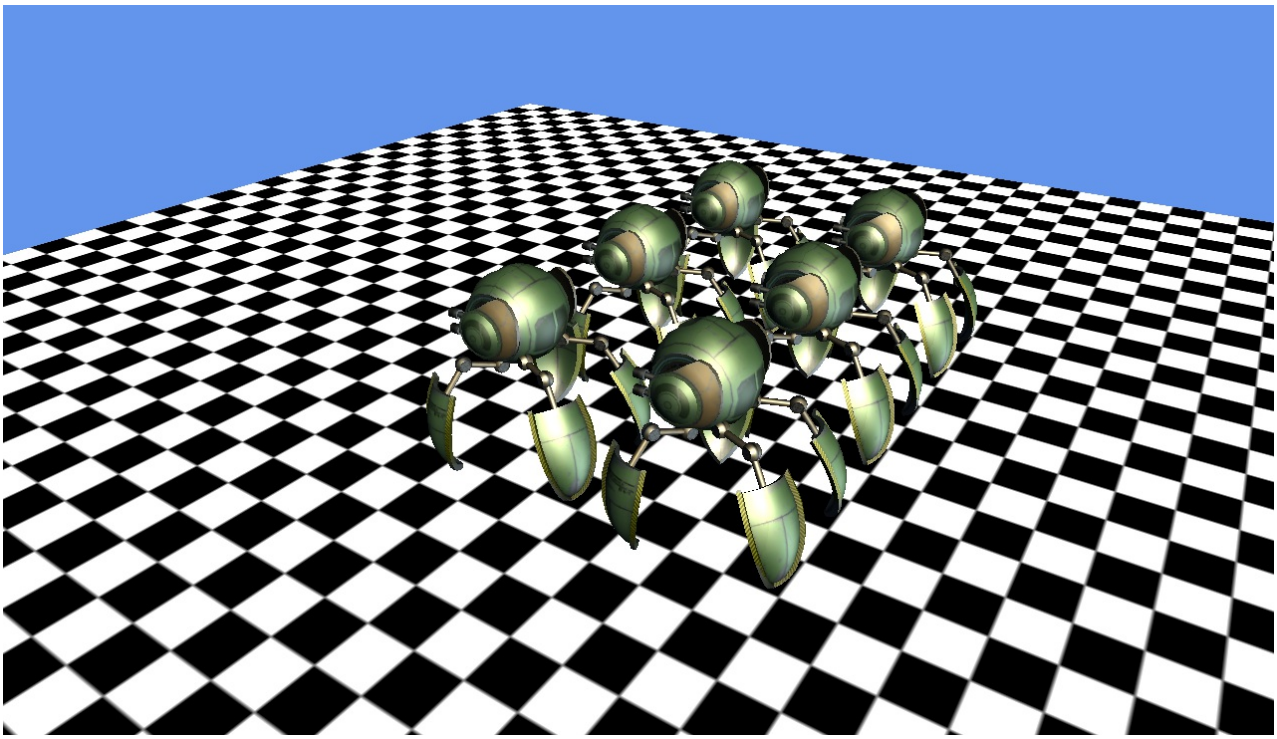
Now if we run the code we can see both the models and the ground at the same time:



If we modify the camera position (such as by increasing its X value, which in this case moves the camera to the left) we can see that the value impacts both the ground and the models:

```
Vector3 cameraPosition = new Vector3(15, 10, 10);
```

This code results in the following:



Summary

This walkthrough showed how to use a vertex array to perform custom rendering. In this case, we created a checkered floor by combining our vertex-based rendering with a texture and `BasicEffect`, but the code presented here serves as the basis for any 3D rendering. We also showed that vertex based rendering can be mixed with models in the same scene.

Related Links

- Checkerboard file (sample)
- Completed project (sample)

3D Coordinates in MonoGame

10/3/2018 • 11 minutes to read • Edit Online

Understanding the 3D coordinate system is an important step in developing 3D games. MonoGame provides a number of classes for positioning, orienting, and scaling objects in 3D space.

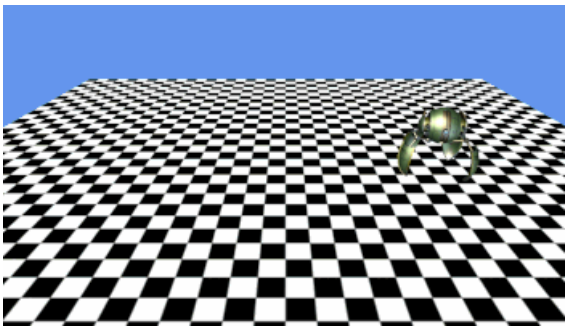
Developing 3D games requires an understanding of how to manipulate objects in a 3D coordinate system. This walkthrough will cover how to manipulate visual objects (specifically a Model). We'll build on the concepts of controlling a model to create a 3D Camera class.

The concepts presented originate from linear algebra, but we'll take a practical approach so that any user without a strong math background can apply these concepts in their own games.

We'll be covering the following topics:

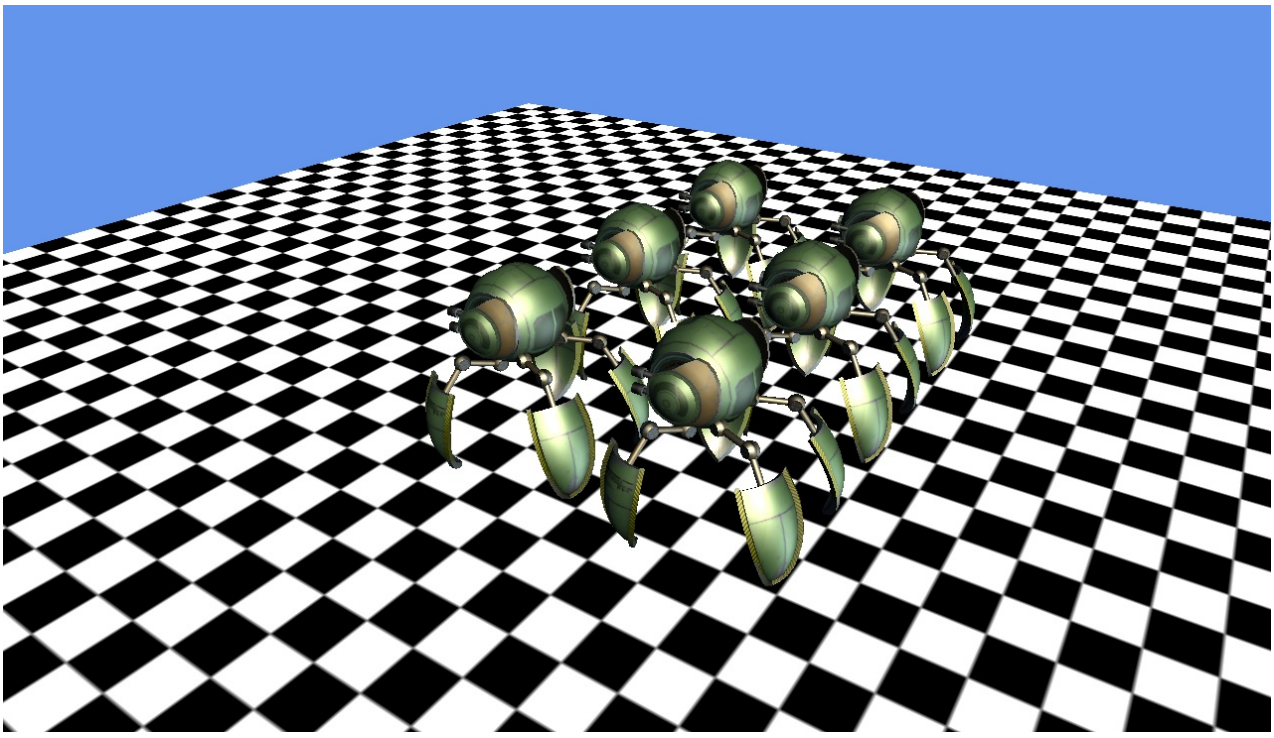
- Creating a Project
- Creating a Robot Entity
- Moving the Robot Entity
- Matrix Multiplication
- Creating the Camera Entity
- Moving the Camera with Input

Once finished, we'll have a project with a robot moving in a circle and a camera which can be controlled by touch input:



Creating a Project

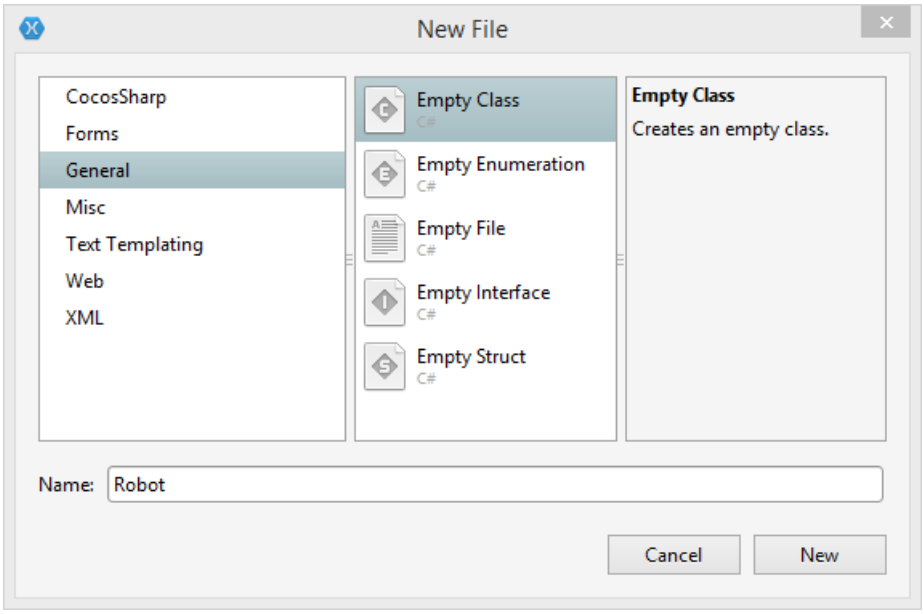
This walkthrough focuses on moving objects in 3D space. We'll begin with the project for rendering models and vertex arrays which can be found here. Once downloaded, unzip and open the project to make sure it runs and we should see the following:



Creating a Robot Entity

Before we begin moving our robot around, we will create a `Robot` class to contain logic for drawing and movement. Game developers refer to this encapsulation of logic and data as an *entity*.

Add a new empty class file to the **MonoGame3D** Portable Class Library (not the platform-specific ModelAndVerts.Android). Name it **Robot** and click **New**:



Modify the `Robot` class as follows:

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

namespace MonoGame3D
{
    public class Robot
    {
        Model model;

        public void Initialize(ContentManager contentManager)
        {
            model = contentManager.Load<Model> ("robot");

        }

        // For now we'll take these values in, eventually we'll
        // take a Camera object
        public void Draw(Vector3 cameraPosition, float aspectRatio)
        {
            foreach (var mesh in model.Meshes)
            {
                foreach (BasicEffect effect in mesh.Effects)
                {
                    effect.EnableDefaultLighting ();
                    effect.PreferPerPixelLighting = true;

                    effect.World = Matrix.Identity;
                    var cameraLookAtVector = Vector3.Zero;
                    var cameraUpVector = Vector3.UnitZ;

                    effect.View = Matrix.CreateLookAt (
                        cameraPosition, cameraLookAtVector, cameraUpVector);

                    float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
                    float nearClipPlane = 1;
                    float farClipPlane = 200;

                    effect.Projection = Matrix.CreatePerspectiveFieldOfView(
                        fieldOfView, aspectRatio, nearClipPlane, farClipPlane);


                }

                // Now that we've assigned our properties on the effects we can
                // draw the entire mesh
                mesh.Draw ();
            }
        }
    }
}
```

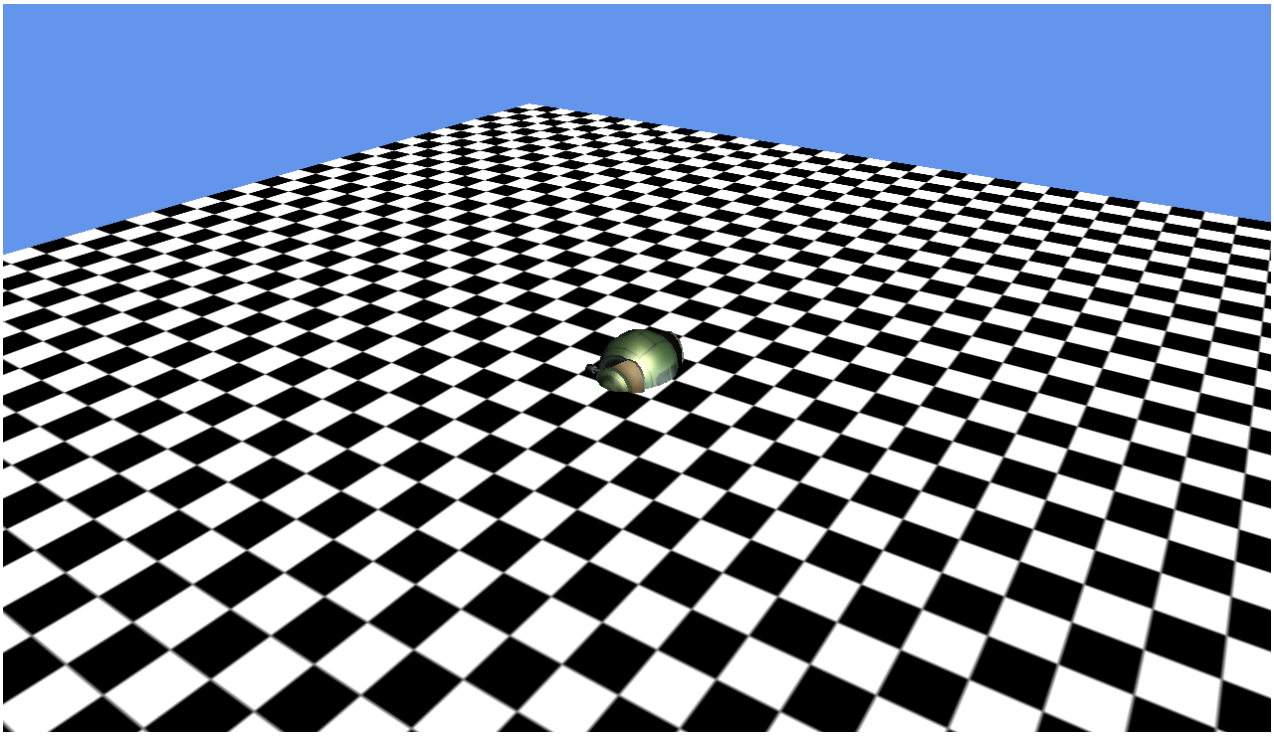
The `Robot` code is essentially the same code in `Game1` for drawing a `Model`. For a review on `Model` loading and drawing, see this guide on working with Models. We can now remove all of the `Model` loading and rendering code from `Game1`, and replace it with a `Robot` instance:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace MonoGame3D
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
```

```
VertexPositionNormalTexture[] floorVerts;

BasicEffect effect;

Texture2D checkerboardTexture;

Vector3 cameraPosition = new Vector3(15, 10, 10);

Robot robot;

public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    graphics.IsFullScreen = true;

    Content.RootDirectory = "Content";
}

protected override void Initialize ()
{
    floorVerts = new VertexPositionNormalTexture[6];

    floorVerts [0].Position = new Vector3 (-20, -20, 0);
    floorVerts [1].Position = new Vector3 (-20,  20, 0);
    floorVerts [2].Position = new Vector3 ( 20, -20, 0);

    floorVerts [3].Position = floorVerts[1].Position;
    floorVerts [4].Position = new Vector3 ( 20,  20, 0);
    floorVerts [5].Position = floorVerts[2].Position;

    int repetitions = 20;

    floorVerts [0].TextureCoordinate = new Vector2 (0, 0);
    floorVerts [1].TextureCoordinate = new Vector2 (0, repetitions);
    floorVerts [2].TextureCoordinate = new Vector2 (repetitions, 0);

    floorVerts [3].TextureCoordinate = floorVerts[1].TextureCoordinate;
    floorVerts [4].TextureCoordinate = new Vector2 (repetitions, repetitions);
    floorVerts [5].TextureCoordinate = floorVerts[2].TextureCoordinate;

    effect = new BasicEffect (graphics.GraphicsDevice);

    robot = new Robot ();
    robot.Initialize (Content);

    base.Initialize ();
}

protected override void LoadContent()
{
    using (var stream = TitleContainer.OpenStream ("Content/checkerboard.png"))
    {
        checkerboardTexture = Texture2D.FromStream (this.GraphicsDevice, stream);
    }
}

protected override void Update(GameTime gameTime)
{
    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    DrawGround ();

    float aspectRatio =
```

```
                graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight;
            robot.Draw (cameraPosition, aspectRatio);

            base.Draw(gameTime);
        }

        void DrawGround()
        {
            var cameraLookAtVector = Vector3.Zero;
            var cameraUpVector = Vector3.UnitZ;

            effect.View = Matrix.CreateLookAt (
                cameraPosition, cameraLookAtVector, cameraUpVector);

            float aspectRatio =
                graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight;
            float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
            float nearClipPlane = 1;
            float farClipPlane = 200;

            effect.Projection = Matrix.CreatePerspectiveFieldOfView(
                fieldOfView, aspectRatio, nearClipPlane, farClipPlane);

            effect.TextureEnabled = true;
            effect.Texture = checkerboardTexture;

            foreach (var pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply ();

                graphics.GraphicsDevice.DrawUserPrimitives (
                            PrimitiveType.TriangleList,
                    floorVerts,
                    0,
                    2);
            }
        }
    }
}
```

If we run the code now we will have a scene with only one robot which is drawn mostly under the floor:



Moving the Robot

Now that we have a `Robot` class, we can add movement logic to the robot. In this case, we'll simply make the robot move in a circle according to the game time. This is a somewhat impractical implementation for a real game since a character may typically respond to input or artificial intelligence, but it provides an environment for us to explore 3D positioning and rotation.

The only information we'll need from outside of the `Robot` class is the current game time. We'll add an `Update` method which will take a `GameTime` parameter. This `GameTime` parameter will be used to increment an angle variable that we'll use to determine the final position for the robot.

First, we'll add the angle field to the `Robot` class under the `model` field:

```
public class Robot
{
    public Model model;

    // new code:
    float angle;
    ...
```

Now we can increment this value in an `Update` function:

```
public void Update(GameTime gameTime)
{
    // TotalSeconds is a double so we need to cast to float
    angle += (float)gameTime.ElapsedGameTime.TotalSeconds;
}
```

We need to make sure that the `Update` method is called from `Game1.Update`:

```
protected override void Update(GameTime gameTime)
{
    robot.Update (gameTime);
    base.Update(gameTime);
}
```

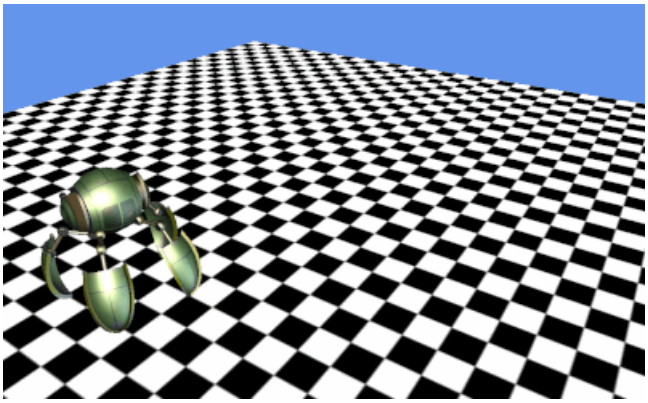
Of course, at this point the angle field does nothing – we need to write code to use it. We'll modify the `Draw` method so that we can calculate the world `Matrix` in a dedicated method:

```
public void Draw(Vector3 cameraPosition, float aspectRatio)
{
    foreach (var mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting ();
            effect.PreferPerPixelLighting = true;
            // We'll be doing our calculations here...
            effect.World = GetWorldMatrix();

            var cameraLookAtVector = Vector3.Zero;
            var cameraUpVector = Vector3.UnitZ;

            effect.View = Matrix.CreateLookAt (
                cameraPosition, cameraLookAtVector, cameraUpVector);

            float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
            float nearClipPlane = 1;
            float farClipPlane = 200;

            effect.Projection = Matrix.CreatePerspectiveFieldOfView(
                fieldOfView, aspectRatio, nearClipPlane, farClipPlane);
        }

        mesh.Draw ();
    }
}
```

Next, we'll implement the `GetWorldMatrix` method in the `Robot` class:

```
Matrix GetWorldMatrix()
{
    const float circleRadius = 8;
    const float heightOffGround = 3;

    // this matrix moves the model "out" from the origin
    Matrix translationMatrix = Matrix.CreateTranslation (
        circleRadius, 0, heightOffGround);

    // this matrix rotates everything around the origin
    Matrix rotationMatrix = Matrix.CreateRotationZ (angle);

    // We combine the two to have the model move in a circle:
    Matrix combined = translationMatrix * rotationMatrix;

    return combined;
}
```

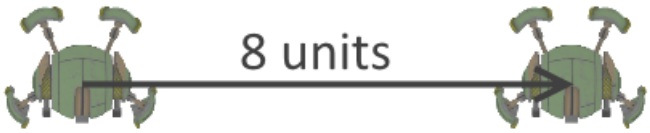
The result of running this code results in the robot moving in a circle:



Matrix Multiplication

The code above rotates the robot by creating a `Matrix` in the `GetWorldMatrix` method. The `Matrix` struct contains 16 float values which can be used to translate (set position), rotate, and scale (set size). When we assign the `effect.World` property, we are telling the underlying rendering system how to position, size, and orient whatever we happen to be drawing (a `Model` or geometry from vertices).

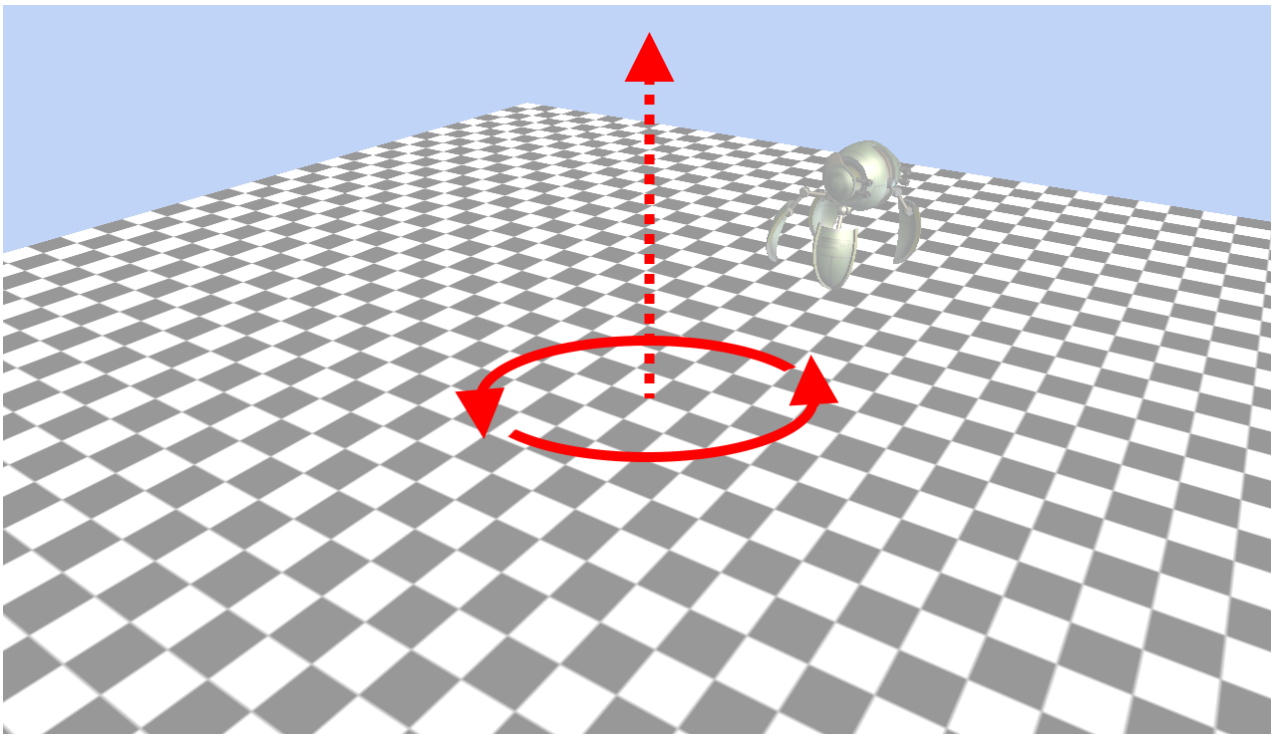
Fortunately, the `Matrix` struct includes a number of methods which simplify the creation of common types of matrices. The first used in the code above is `Matrix.CreateTranslation`. The mathematical term *translation* refers to an operation which results in a point (or in our case a model) moving from one location to another without any other modification (such as rotating or resizing). The function takes an X, Y, and Z value for translation. If we view our scene from top-down, our `CreateTranslation` method (in isolation) performs the following:



The second matrix that we create is a rotation matrix using the `CreateRotationZ` matrix. This is one of three methods which can be used to create rotation:

- `CreateRotationX`
- `CreateRoationY`
- `CreateRotationZ`

Each method creates a rotation matrix by rotating about a given axis. In our case, we are rotating about the Z axis, which points "up". The following can help visualize how axis-based rotation works:

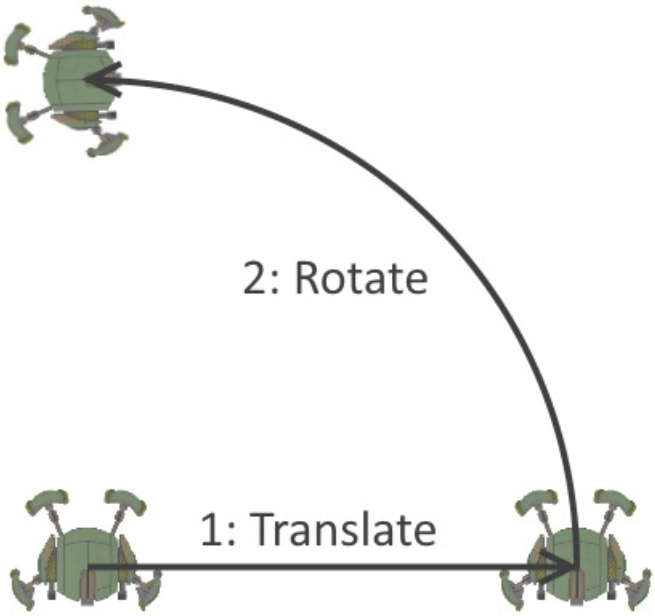


We are also using the `CreateRotationZ` method with the angle field, which increments over time due to our `Update` method being called. The result is that the `CreateRotationZ` method causes our robot to orbit around the origin as time passes.

The final line of code combines the two matrices into one:

```
Matrix combined = translationMatrix * rotationMatrix;
```

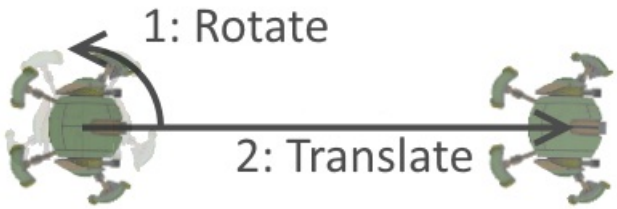
This is referred to as matrix multiplication, which works slightly different than regular multiplication. The *commutative property of multiplication* states that the order of numbers in a multiplication operation does not change the result. That is, 3 * 4 is equivalent to 4 * 3. Matrix multiplication differs in that it is not commutative. That is, the above line can be read as "Apply the translationMatrix to move the model, then rotate everything by applying the rotationMatrix". We could visualize the way that the above line affects the position and rotation as follows:



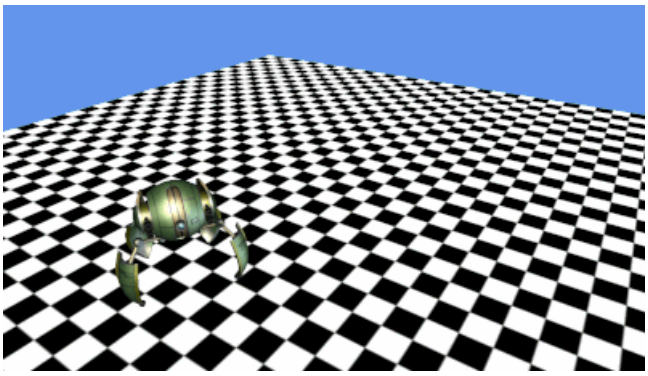
To help understand how the order of matrix multiplication can impact the outcome, consider the following, where the matrix multiplication is inverted:

```
Matrix combined = rotationMatrix * translationMatrix;
```

The code above would first rotate the model in-place, then translate it:



If we run the code with the inverted multiplication, we'll notice that since the rotation applies first, it only impacts the orientation of the model and the position of the model stays the same. In other words, the model rotates in place:



Creating the Camera Entity

The `Camera` entity will contain all of the logic necessary to perform input-based movement and to provide properties for assigning properties on the `BasicEffect` class.

First we'll implement a static camera (no input-based movement) and integrate it into our existing project. Add a new class to the **MonoGame3D** Portable Class Library (the same project with `Robot.cs`) and name it **Camera**. Replace the contents of the file with the following code:

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

namespace MonoGame3D
{
    public class Camera
    {
        // We need this to calculate the aspectRatio
        // in the ProjectionMatrix property.
        GraphicsDevice graphicsDevice;

        Vector3 position = new Vector3(15, 10, 10);

        public Matrix ViewMatrix
        {
            get
            {
                var lookAtVector = Vector3.Zero;
                var upVector = Vector3.UnitZ;

                return Matrix.CreateLookAt (
                    position, lookAtVector, upVector);
            }
        }

        public Matrix ProjectionMatrix
        {
            get
            {
                float fieldOfView = Microsoft.Xna.Framework.MathHelper.PiOver4;
                float nearClipPlane = 1;
                float farClipPlane = 200;
                float aspectRatio = graphicsDevice.Viewport.Width / (float)graphicsDevice.Viewport.Height;

                return Matrix.CreatePerspectiveFieldOfView(
                    fieldOfView, aspectRatio, nearClipPlane, farClipPlane);
            }
        }

        public Camera(GraphicsDevice graphicsDevice)
        {
            this.graphicsDevice = graphicsDevice;
        }

        public void Update(GameTime gameTime)
        {
            // We'll be doing some input-based movement here
        }
    }
}
```

The code above is very similar to the code from `Game1` and `Robot` which assign the matrices on `BasicEffect`.

Now we can integrate the new `Camera` class into our existing projects. First, we'll modify the `Robot` class to take a `Camera` instance in its `Draw` method, which will eliminate a lot of duplicate code. Replace the `Robot.Draw` method with the following:

```
public void Draw(Camera camera)
{
    foreach (var mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting ();
            effect.PreferPerPixelLighting = true;

            effect.World = GetWorldMatrix();
            effect.View = camera.ViewMatrix;
            effect.Projection = camera.ProjectionMatrix;
        }

        mesh.Draw ();
    }
}
```

Next ,modify the `Game1.cs` file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace MonoGame3D
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;

        VertexPositionNormalTexture[] floorVerts;

        BasicEffect effect;

        Texture2D checkerboardTexture;

        // New camera code
        Camera camera;

        Robot robot;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            graphics.IsFullScreen = true;

            Content.RootDirectory = "Content";
        }

        protected override void Initialize ()
        {
            floorVerts = new VertexPositionNormalTexture[6];

            floorVerts [0].Position = new Vector3 (-20, -20, 0);
            floorVerts [1].Position = new Vector3 (-20,  20, 0);
            floorVerts [2].Position = new Vector3 ( 20, -20, 0);

            floorVerts [3].Position = floorVerts[1].Position;
            floorVerts [4].Position = new Vector3 ( 20,  20, 0);
            floorVerts [5].Position = floorVerts[2].Position;

            int repetitions = 20;

            floorVerts [0].TextureCoordinate = new Vector2 (0, 0);
            floorVerts [1].TextureCoordinate = new Vector2 (0, repetitions);
            floorVerts [2].TextureCoordinate = new Vector2 (repetitions, 0);
```

```
            floorVerts [3].TextureCoordinate = floorVerts[1].TextureCoordinate;
            floorVerts [4].TextureCoordinate = new Vector2 (repetitions, repetitions);
            floorVerts [5].TextureCoordinate = floorVerts[2].TextureCoordinate;

            effect = new BasicEffect (graphics.GraphicsDevice);

            robot = new Robot ();
            robot.Initialize (Content);

            // New camera code
            camera = new Camera (graphics.GraphicsDevice);

            base.Initialize ();
        }

        protected override void LoadContent()
        {
            using (var stream = TitleContainer.OpenStream ("Content/checkerboard.png"))
            {
                checkerboardTexture = Texture2D.FromStream (this.GraphicsDevice, stream);
            }
        }

        protected override void Update(GameTime gameTime)
        {
            robot.Update (gameTime);
            // New camera code
            camera.Update (gameTime);
            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            DrawGround ();

            // New camera code
            robot.Draw (camera);

            base.Draw(gameTime);
        }

        void DrawGround()
        {
            // New camera code
            effect.View = camera.ViewMatrix;
            effect.Projection = camera.ProjectionMatrix;

            effect.TextureEnabled = true;
            effect.Texture = checkerboardTexture;

            foreach (var pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply ();

                graphics.GraphicsDevice.DrawUserPrimitives (
                            PrimitiveType.TriangleList,
                    floorVerts,
                    0,
                    2);
            }
        }
    }
}
```

The modifications to the `Game1` from the previous version (which are identified with `// New camera code`) are:

- `Camera` field in `Game1`
- `Camera` instantiation in `Game1.Initialize`
- `Camera.Update` call in `Game1.Update`
- `Robot.Draw` now takes a `Camera` parameter
- `Game1.Draw` now uses `Camera.ViewMatrix` and `Camera.ProjectionMatrix`

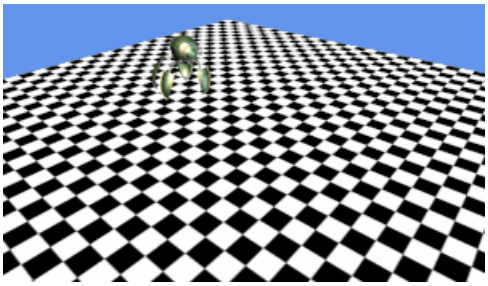
Moving the Camera with Input

So far, we've added a `Camera` entity but haven't done anything with it to change runtime behavior. We will add behavior which allows the user to:

- Touch the left side of the screen to turn the camera toward the left
- Touch the right side of the screen to turn the camera to the right
- Touch the center of the screen to move the camera forward

Making lookAt Relative

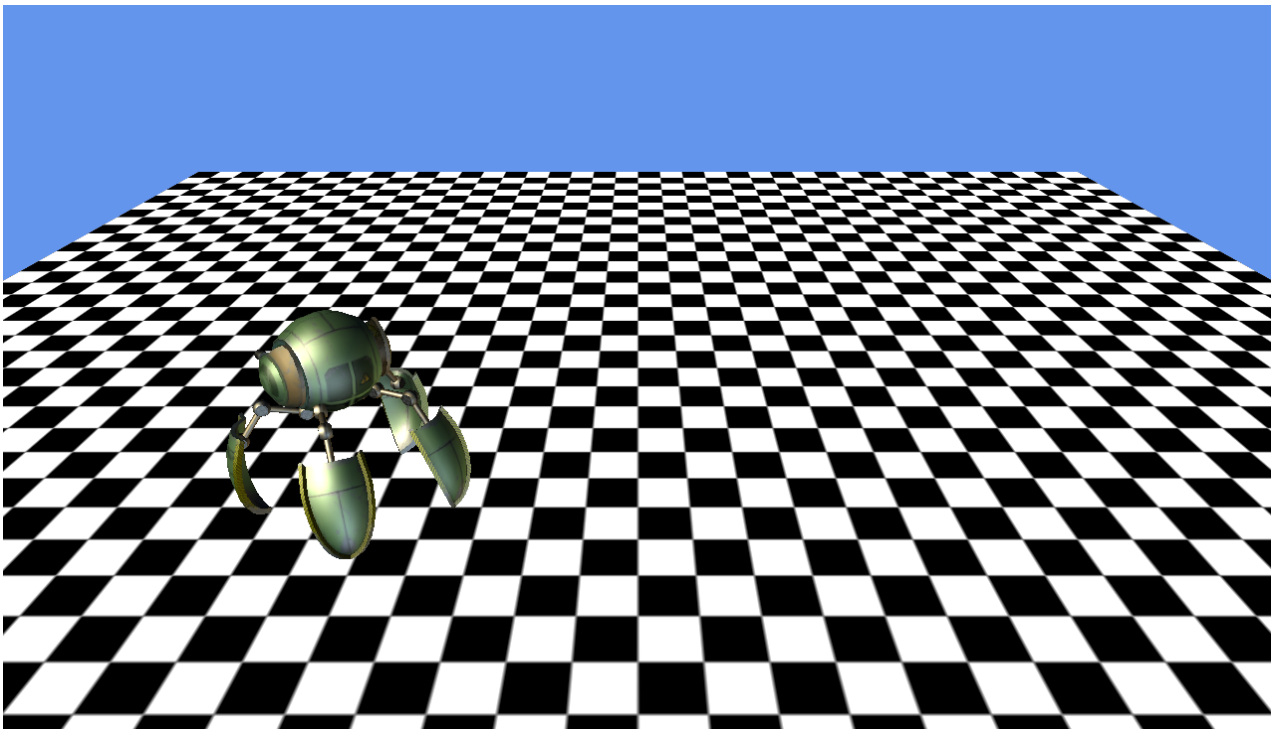
First we'll update the `Camera` class to include an `angle` field which will be used to set the direction that the `Camera` is facing. Currently, our `Camera` determines the direction it is facing through the local `lookAtVector`, which is assigned to `Vector3.Zero`. In other words, our `Camera` always looks at the origin. If the Camera moves, then the angle that the camera is facing will also change:



We want the `Camera` to be facing the same direction regardless of its position – at least until we implement the logic for rotating the `Camera` using input. The first change will be to adjust the `lookAtVector` variable to be based off of our current location, rather than looking at an absolute position:

```
public class Camera
{
    GraphicsDevice graphicsDevice;

    // Let's start at X = 0 so we're looking at things head-on
    Vector3 position = new Vector3(0, 20, 10);

    public Matrix ViewMatrix
    {
        get
        {
            var lookAtVector = new Vector3 (0, -1, -.5f);
            lookAtVector += position;

            var upVector = Vector3.UnitZ;

            return  Matrix.CreateLookAt (
                position, lookAtVector, upVector);
        }
    }
    ...
```

This results in the `Camera` viewing the world straight-on. Notice that the initial `position` value has been modified to `(0, 20, 10)` so the `Camera` is centered on the X axis. Running the game displays:



Creating an angle Variable

The `lookAtVector` variable controls the angle that our camera is viewing. Currently it is fixed to view down the negative Y axis, and slightly tilted down (from the `-.5f` Z value). We'll create an `angle` variable which will be used to adjust the `lookAtVector` property.

In earlier sections of this walkthrough we showed that matrices can be used to rotate how objects are drawn. We can also use matrices to rotate vectors like the `lookAtVector` using the `Vector3.Transform` method.

Add an `angle` field and modify the `ViewMatrix` property as follows:

```
public class Camera
{
    GraphicsDevice graphicsDevice;

    Vector3 position = new Vector3(0, 20, 10);

    float angle;

    public Matrix ViewMatrix
    {
        get
        {
            var lookAtVector = new Vector3 (0, -1, -.5f);
            // We'll create a rotation matrix using our angle
            var rotationMatrix = Matrix.CreateRotationZ (angle);
            // Then we'll modify the vector using this matrix:
            lookAtVector = Vector3.Transform (lookAtVector, rotationMatrix);
            lookAtVector += position;

            var upVector = Vector3.UnitZ;

            return  Matrix.CreateLookAt (
                position, lookAtVector, upVector);
        }
    }
    ...
```

Reading input

Our `Camera` entity can now be fully controlled through its position and angle variables – we just need to change

them according to input.

First, we'll get the `TouchPanel` state to find where the user is touching the screen. For more information on using the `TouchPanel` class, see the TouchPanel API reference.

If the user is touching on the left third then we'll adjust the `angle` value so the `Camera` rotates left, and if the user is touching on the right third, we'll rotate the other way. If the user is touching in the middle third of the screen, then we'll move the `Camera` forward.

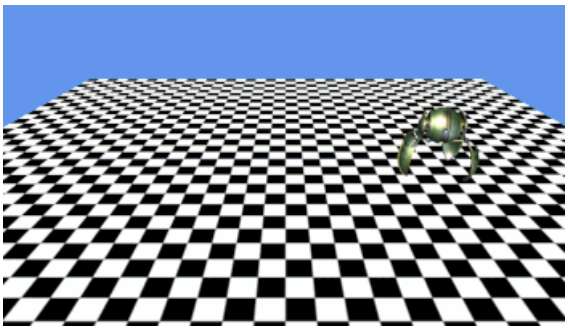
First, add a using statement to qualify the `TouchPanel` and `TouchCollection` classes in `Camera.cs`:

```
using Microsoft.Xna.Framework.Input.Touch;
```

Next, modify the `Update` method to read the touch panel and adjust the `angle` and `position` variables appropriately:

```
public void Update(GameTime gameTime)
{
    TouchCollection touchCollection = TouchPanel.GetState();

    bool isTouchingScreen = touchCollection.Count > 0;
    if (isTouchingScreen)
    {
        var xPosition = touchCollection [0].Position.X;

        float xRatio = xPosition / (float)graphicsDevice.Viewport.Width;

        if (xRatio < 1 / 3.0f)
        {
            angle += (float)gameTime.ElapsedGameTime.TotalSeconds;
        }
        else if (xRatio < 2 / 3.0f)
        {
            var forwardVector = new Vector3 (0, -1, 0);

            var rotationMatrix = Matrix.CreateRotationZ (angle);
            forwardVector = Vector3.Transform (forwardVector, rotationMatrix);

            const float unitsPerSecond = 3;

            this.position += forwardVector * unitsPerSecond *
                (float)gameTime.ElapsedGameTime.TotalSeconds ;
        }
        else
        {
            angle -= (float)gameTime.ElapsedGameTime.TotalSeconds;
        }
    }
}
```

Now the `Camera` will respond to touch input:



The Update method begins by calling `TouchPanel.GetState`, which returns a collection of touches. Although `TouchPanel.GetState` can return multiple touch points, we'll only worry about the first one for simplicity.

If the user is touching the screen, then the code checks to see if the first touch is in the left, middle, or right third of the screen. The left and right thirds rotate the camera by increasing or decreasing the `angle` variable according to the `TotalSeconds` value (so that the game behaves the same regardless of frame rate).

If the user is touching the center third of the screen, then the camera will move forward. This is accomplished first by obtaining the forward vector, which is initially defined as pointing towards the negative Y axis, then rotated by a matrix created using `Matrix.CreateRotationZ` and the `angle` value. Finally the `forwardVector` is applied to `position` using the `unitsPerSecond` coefficient.

Summary

This walkthrough covers how to move and rotate `Models` in 3D space using `Matrices` and the `BasicEffect.World` property. This form of movement provides the basis for moving objects in 3D games. This walkthrough also covers how to implement a `Camera` entity for viewing the world from any position and angle.

Related Links

- MonoGame API Link
- Finished Project (sample)

MonoGame GamePad Reference

10/3/2018 • 3 minutes to read • Edit Online

GamePad is a standard, cross-platform class for accessing input devices in MonoGame.

`GamePad` can be used to read input from input devices on multiple MonoGame platforms. This guide shows how to work with the GamePad class. Since each input device differs in layout and the number of buttons it provides, this guide includes diagrams that show the various device mappings.

GamePad as a Replacement for Xbox360GamePad

The original XNA API provided the `Xbox360GamePad` class for reading input from a game controller on the Xbox 360 or PC. MonoGame has replaced this with a `GamePad` class since Xbox 360 controllers cannot be used on most MonoGame platforms (such as iOS or Xbox One). Despite the name change, the usage of the `GamePad` class is similar to the `Xbox360GamePad` class.

Reading Input from GamePad

The `GamePad` class provides a standardized way of reading input on any MonoGame platform. It provides information through two methods:

- `GetState` – returns the current state of the controller's buttons, analog sticks, and d-pad.
- `GetCapabilities` – returns information about the capabilities of hardware, such as whether the controller has certain buttons or supports vibration.

Example: Moving a Character

The following code shows how the left thumb stick can be used to move a character by setting its `XVelocity` and `YVelocity` properties. This code assumes that `characterInstance` is an instance of an object which has `XVelocity` and `YVelocity` properties:

```
// In Update, or some code called every frame:
var gamePadState = GamePad.GetState(PlayerIndex.One);
// Use gamePadState to move the character
characterInstance.XVelocity = gamePadState.ThumbSticks.Left.X * characterInstance.MaxSpeed;
characterInstance.YVelocity = gamePadState.ThumbSticks.Left.Y * characterInstance.MaxSpeed;
```

Example: Detecting Pushes

`GamePadState` provides information about the current state of the controller, such as whether a certain button is pressed. Certain actions, such as making a character jump, require checking if the button was pushed (was not down last frame, but is down this frame) or released (was down last frame, but not down this frame).

To perform this type of logic, local variables that store the previous frame's `GamePadState` and the current frame's `GamePadState` must be created. The following example shows how to store and use the previous frame's `GamePadState` to implement jumping:

```
// At class scope:
// Store the last frame's and this frame's GamePadStates.
// "new" them so that code doesn't have to perform null checks:
GamePadState lastFrameGamePadState = new GamePadState();
GamePadState currentGamePadState = new GamePadState();
protected override void Update(GameTime gameTime)
{
    // store off the last state before reading the new one:
    lastFrameGamePadState = currentGamePadState;
    currentGamePadState = GamePad.GetState(PlayerIndex.One);
    bool wasAButtonPushed =
currentGamePadState.Buttons.A == ButtonState.Pressed
        && lastFrameGamePadState.Buttons.A == ButtonState.Released;
    if(wasAButtonPushed)
    {
        MakeCharacterJump();
    }
...
}
```

Example: Checking for Buttons

`GetCapabilities` can be used to check if a controller has certain hardware, such as a particular button or analog stick. The following code shows how to check for the B and Y buttons on a controller in a game which requires the presence of both buttons:

```
var capabilities = GamePad.GetCapabilities(PlayerIndex.One);
bool hasBButton = capabilities.HasBButton;
bool hasXButton = capabilities.HasXButton;
if(!hasBButton || !hasXButton)
{
    NotifyUserOfMissingButtons();
}
```

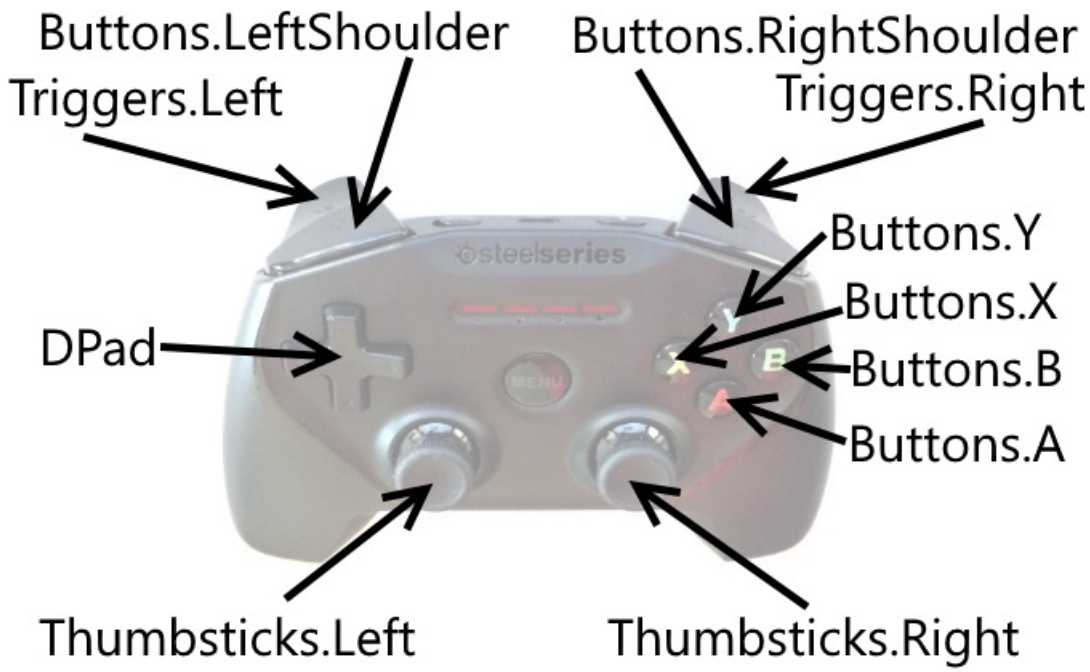
iOS

iOS apps support wireless game controller input.

> **IMPORTANT**
> The NuGet packages for MonoGame 3.5 do not include support for wireless game controllers. Using the GamePad class on iOS requires building MonoGame 3.5 from source or using the MonoGame 3.6 NuGet binaries.

iOS Game Controller

The `GamePad` class returns properties read from wireless controllers. The properties in the `GamePad` provide good coverage for the standard iOS controller hardware, as shown in the following diagram:



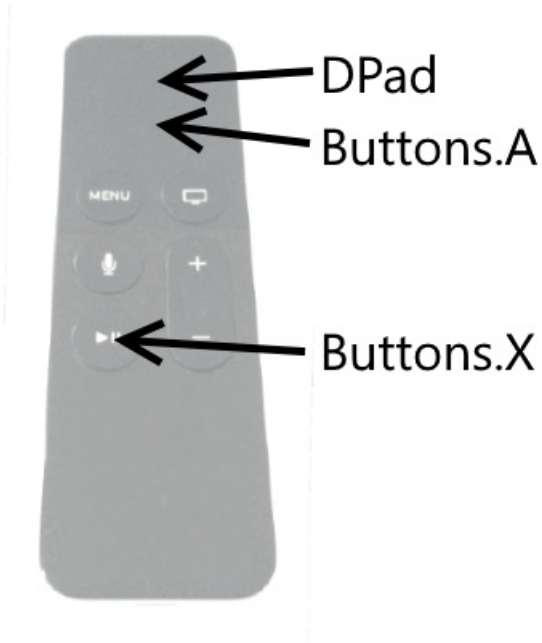
Apple TV

Apple TV games can use the Siri Remote or wireless game controllers for input.

Siri Remote

Siri Remote is the native input device for Apple TV. Although values from the Siri Remote can be read through events (as shown in the Siri Remote and Bluetooth Controllers guide), the `GamePad` class can return values from the Siri Remote.

Notice that `GamePad` can only read input from the play button and touch surface:



Since the touch surface movement is read through the `DPad` property, movement values are reported using the `ButtonState` class. In other words, values are only available as `ButtonState.Pressed` or `ButtonState.Released`, as opposed to numerical values or gestures.

Apple TV Game Controller

Game controllers for Apple TV behave identically to game controllers for iOS apps. For more information, see the

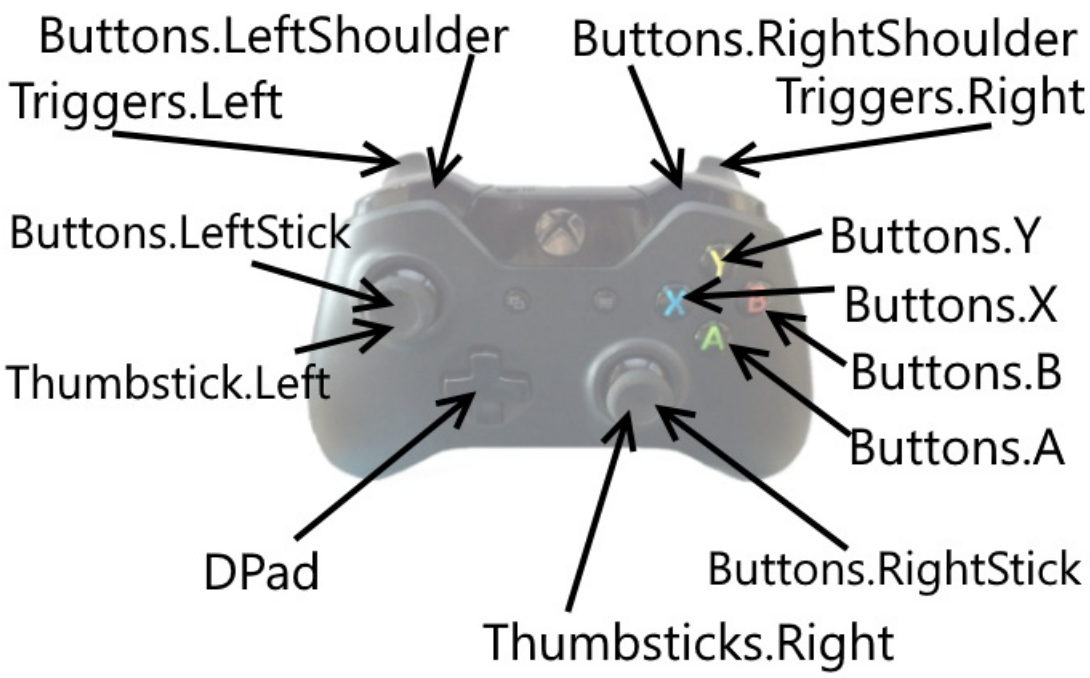
iOS Game Controller section.

Xbox One

The Xbox One console supports reading input from an Xbox One game controller.

Xbox One Game Controller

The Xbox One game controller is the most common input device for the Xbox One. The `GamePad` class provides input values from the game controller hardware.



Summary

This guide provided an overview of MonoGame's `GamePad` class, how to implement input-reading logic, and diagrams of common `GamePad` implementations.

Related Links

- MonoGame GamePad

MonoGame Platform Specific Considerations

10/3/2018 • 2 minutes to read • Edit Online

MonoGame on Universal Windows Platform (UWP)

This walkthrough covers MonoGame Universal Windows Platform (UWP) project creation and content loading. UWP apps can run on all Windows 10 devices, including desktops, tablets, Windows Phones, and Xbox One.

Creating a MonoGame UWP project

10/3/2018 • 3 minutes to read • Edit Online

MonoGame can be used to create games and apps for Universal Windows Platform, targeting multiple devices with one codebase and one set of content.

This walkthrough covers MonoGame Universal Windows Platform (UWP) project creation and content loading. UWP apps can run on all Windows 10 devices, including desktops, tablets, Windows Phones, and Xbox One.

This walkthrough creates an empty project which displays a *cornflower blue* background (the traditional background color of XNA apps).

Requirements

Developing MonoGame UWP apps requires:

- Windows 10 operating system
- Any version of Visual Studio 2015
- Windows 10 developer tools
- Setting device to developer mode
- MonoGame 3.5 for Visual Studio or newer

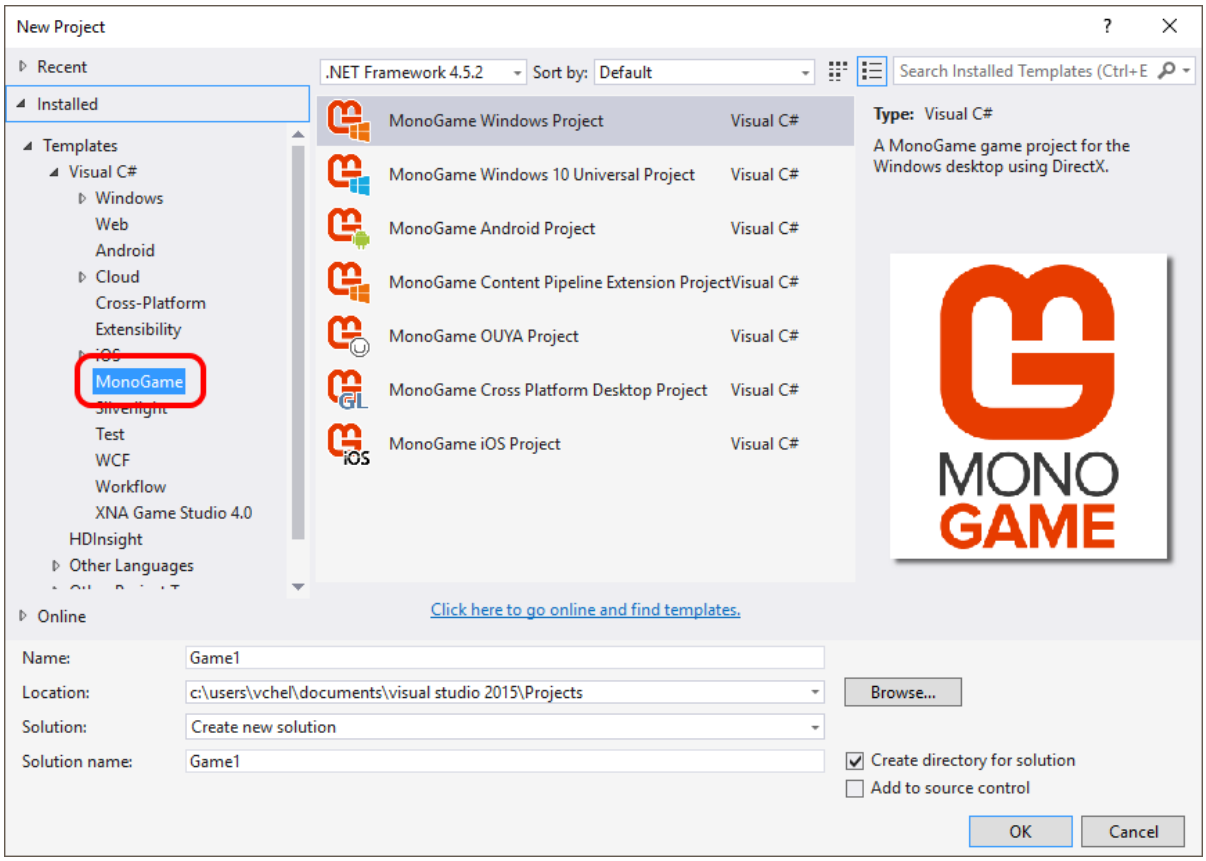
For more information, see this page on setting up for Windows 10 UWP development.

Xbox One games can be developed on retail Xbox One hardware. Additional software is required on both the developing PC and the Xbox One. For information on configuring an Xbox One for game development, see this page on setting up an Xbox One.

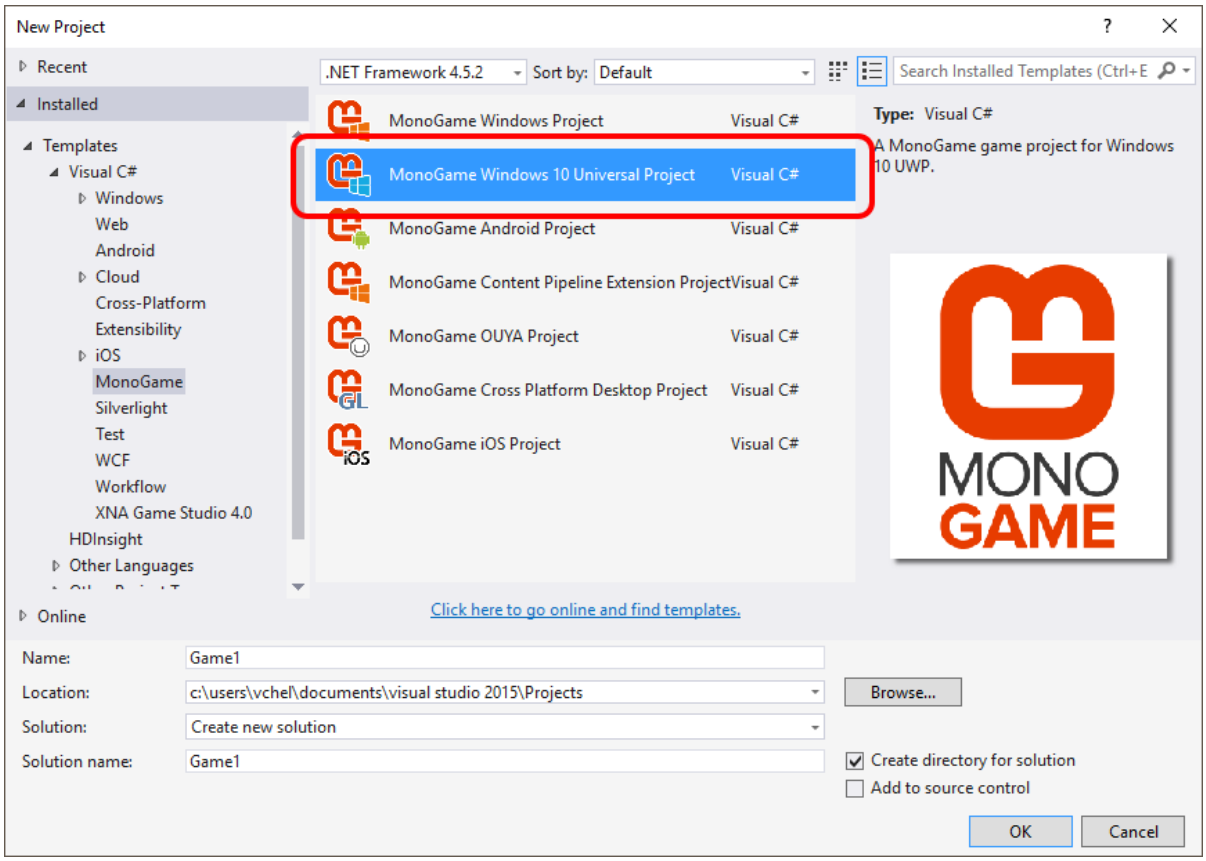
Creating an Empty Template

Once all required resources have been installed and developer mode has been enabled on the Windows 10 machine, we can create a new MonoGame project using Visual Studio by following these steps:

1. Select **File** > **New** > **Project...**
2. Select the **Installed** > **Templates** > **Visual C#** > **MonoGame** category:

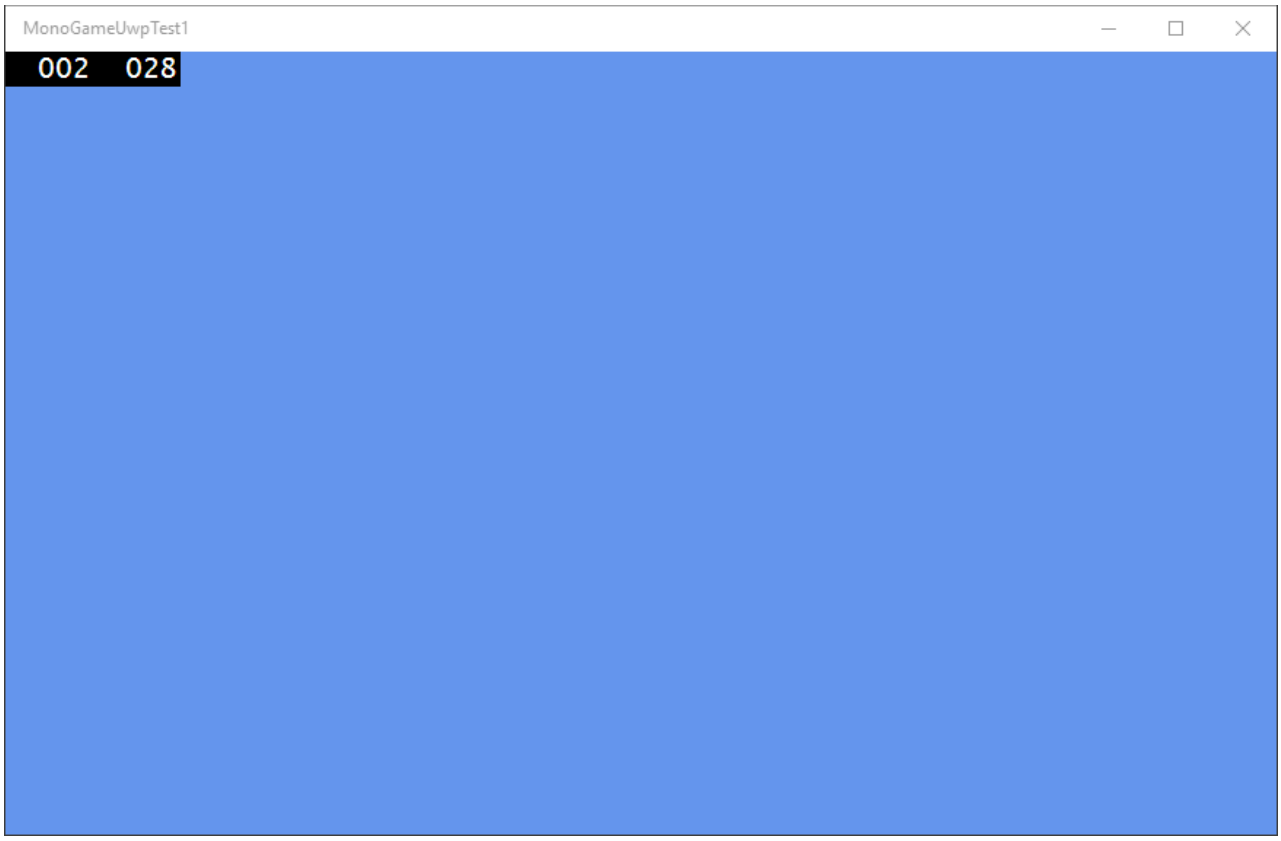


3. Select the **MonoGame Windows 10 Universal Project** option:



4. Enter a name for the new project and click **OK**. If Visual Studio displays any errors after clicking OK, verify that Windows 10 tools are installed and that the device is in developer mode.

Once Visual Studio finishes creating the template, we can run it to see the empty project running:

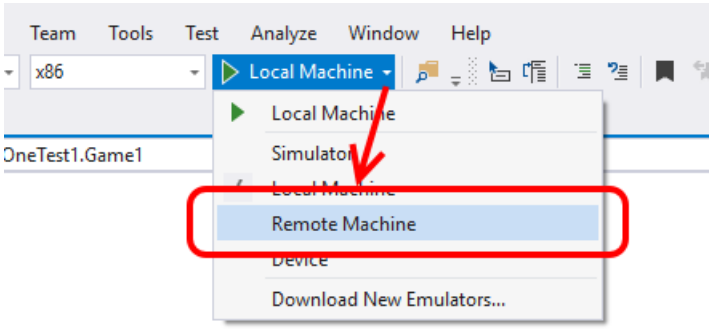


The numbers in the corners provide diagnostic information. This information can be removed by deleting the code in `App.xaml.cs` in the `DEBUG` block in the `OnLaunched` method:

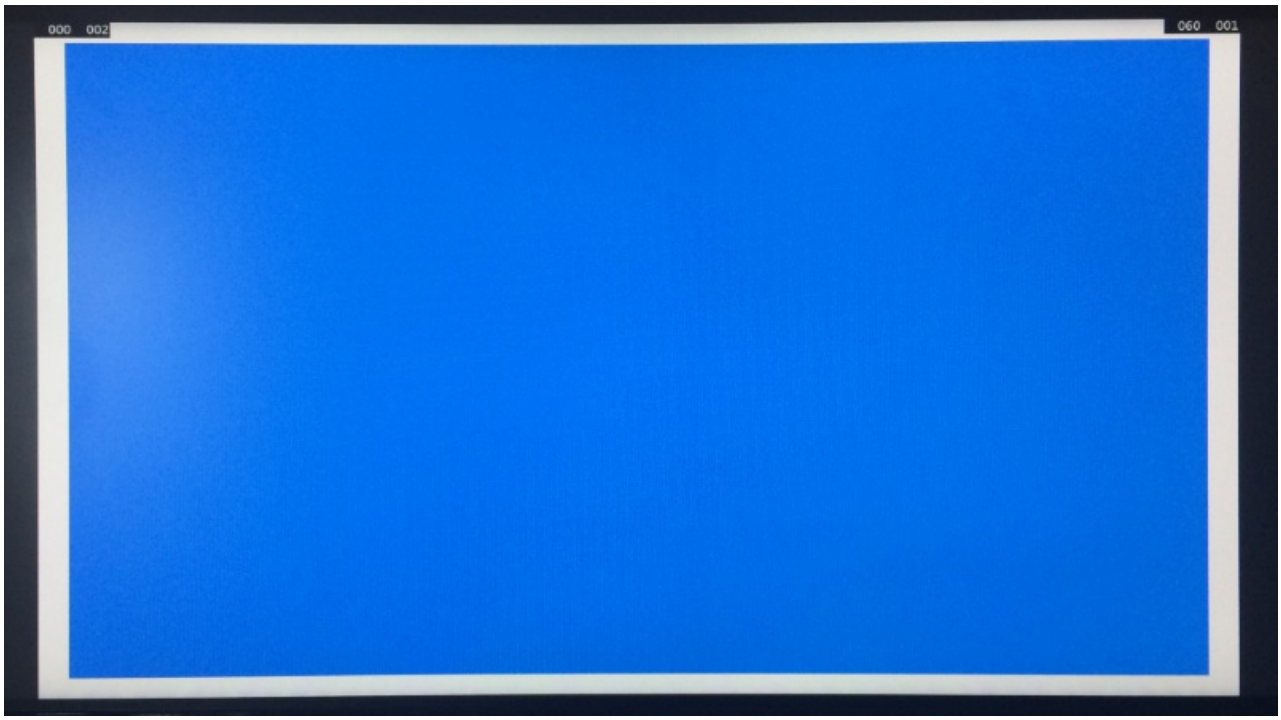
```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
#if DEBUG
    if (System.Diagnostics.Debugger.IsAttached)
    {
        this.DebugSettings.EnableFrameRateCounter = true;
    }
#endif
    ...
```

Running on Xbox One

UWP projects can deploy to any Windows 10 device from the same project. After setting up the Windows 10 development machine and the Xbox One, UWP apps can be deployed by switching the target to Remote Machine and entering the Xbox One's IP address:



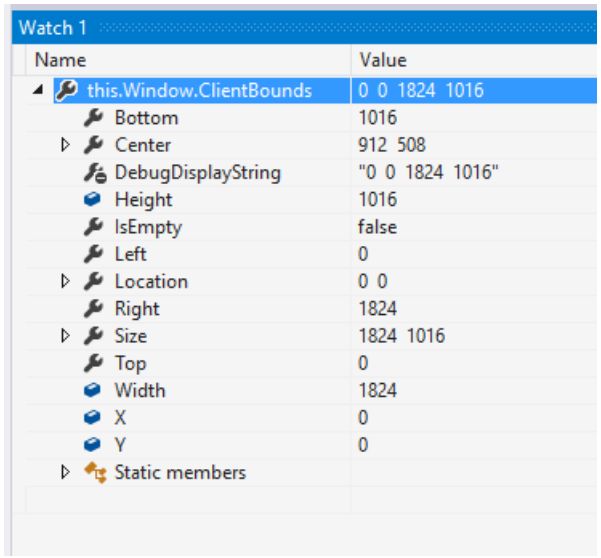
On Xbox One, the white border represents the non-safe area for TVs. For more information, see the safe area section.



Safe Area on Xbox One

Developing games for consoles requires considering the safe area, which is an area in the center of the screen which should contain all critical visuals (such as UI or HUD). The area outside of the safe area is not guaranteed to be visible on all televisions, so visuals placed in this area may be partly or fully invisible on some displays.

The MonoGame template for Xbox One considers the safe area and renders it as a white border. This area is also reflected at runtime in the game's `Window.ClientBounds` property as shown in this image of the watch window in Visual Studio. Notice that the height of the client bounds is 1016, despite 1920x1080 display resolution:



Referencing Content in UWP Projects

Content in MonoGame projects can be referenced directly from file or through the MonoGame Content Pipeline. Small game projects may benefit from the simplicity of loading from file. Larger projects will benefit from using the content pipeline to optimize content to reduce size and load times. Unlike XNA on the Xbox 360, the `System.IO.File` class is available on Xbox One UWP apps.

For more information on loading content using the content pipeline, see the Content Pipeline Guide.

Loading Content From File

Unlike iOS and Android, UWP projects can reference files relative to the executable. Simple games can use this

technique load content without needing to modify and build the content pipeline project.

To load a `Texture2D` from file:

1. Add a .png file to the Content folder in the UWP project. Adding content to the Content folder is a convention in XNA and MonoGame.
2. Right-click on the newly-added PNG and select Properties.
3. Change the **Copy to Output Directory** to **Copy if Newer**.
4. Add the following code to your game's Initialize method to load a `Texture2D`:

```
Texture2D texture;
using (var stream = System.IO.File.OpenRead("Content/YourPngName.png"))
{
    texture = Texture2D.FromStream(graphics.GraphicsDevice, stream);
}
```

For more information on using a `Texture2D`, see the Intro to MonoGame guide.

Summary

This guide covers how to create a new UWP project and UWP-specific considerations when loading files. Developers who are interested in creating full UWP games can read more about MonoGame in the Introduction to MonoGame Guide.