Nu Game Engine

Bryan Edds, 2014

Table of Contents

What's It All About?

The Nu Game Engine is a **Simple**, **Purely-Functional**(ish), **2d Game Engine** written in **F#**.

Let me explain each of those terms –

Simple

Nu is still young, and so it has just about no frills. Is there a particle or special effects system? Not yet, I'm afraid. Is there a sophisticated animation system? Again, not yet. However, there is a tile map system that utilizes Tiled#, and there is a physics system that utilizes Farseer Physics. Rendering, audio, and other IO systems are handled in a cross-platform way with SDL2 / SDL2#. In addition to that, there is an asset management system to make sure your game can run on memory-constrained devices such as the iPhone. On top of all that, there is a built-in game editor called NuEdit! So while there are plenty of missing features, you can see they might be worth waiting for, or even building for yourself!

Purely-Functional(ish)

Nu is built on immutable types, and unlike with other game engines, data transformations and state transitions are implemented with copying rather than mutation.

Don't mistake Nu for being slow, however. Notice I said Purely-Functional-ish. The 'ish' means that there are some imperative operations going on in Nu, almost entirely behind the scenes. For example, the Farseer physics system is written in an imperative style in C#, and some parts of Nu are optimized with imperative code as well. Fortunately, nearly all of this will be transparent to you as the user. When writing code that utilizes, feel empowered to write in the pure-functional style.

2d Game Engine

Nu is not a code library. It is a game software framework, and thus sets up a specific way of approaching and thinking about the design of 2d games. Of course, Nu is intended to be a broadly generic toolkit for 2d game development, but there are some design choices that may sometimes constrain you as much as they help you. Figure out how to leverage Nu's design for your game. If it's a complete mismatch, it might be time to consider using something else.

F#

We know what F# is, so why use it? First, and foremost, its cross-platformedness. Theoretically, Nu should run fine on Mono for systems such as Android, iOS, OSX, and *nixes. It definitely runs on .NET for Windows. Note my weasel-word "theoretically"; Nu is still in such an early stage that it has yet to be configured, deployed, or tested on Mono. Nonetheless, since Nu only takes dependencies on cross-platform libraries, there should be no reason why it can't with a little bit of appropriate nudging.

But more on why F#. F# is probably the best mainstream language available for writing a cross-platform functional game engine. Unlike Clojure, F#'s static type system makes the code easier to reason about and dare I say more efficient. Unlike JVM languages, F# allows us to code and debug with Visual Studio. Finally, I speculate that game developers have more familiarity with the .NET ecosystem than the JVM, so that leverage is right there.

Getting Started

Nu is made available by a GitHub repository located at https://github.com/bryanedds/FPWorks. To obtain it, first ***fork*** the repository's latest ***release*** to your own GitHub account (register as a new GitHub use if you don't already have an account). Second, ***clone*** the forked repository to your local machine (instructions here https://help.github.com/articles/fork-a-repo). The Nu Game Engine is now yours!

Note: Unlike code libraries that are distributed via NuGet, forking and cloning the FP Works repository at GitHub is how you attain Nu. You will be happy with this once you need to make and debug your own changes to the game engine!

Upon inspecting your clone of the repository, the first thing you might notice about it is that is contains more than just the Nu Game Engine. It also includes the source for the **Aml** programming language, the **Prime** F# code library, the sample game **BlazeVector** (which we'll be studying in this document), and my WIP role-playing game **OmniBlade**. Both Prime and BlazeVector are required to build the BlazeVector solution we'll be opening in this tutorial, and the rest of the stuff is safely ignored.

To open the BlazeVector solution, first make sure to have Visual Studio 2013 installed (or perhaps an earlier version – not tested!) Then navigate to the ./BlazeVector/BlazeVector folder and open the BlazeVector.sln file. Attempt to build the whole solution. If there is a problem with building it, try to figure it out, and failing that, ask me questions via bryanedds@gmail.com.

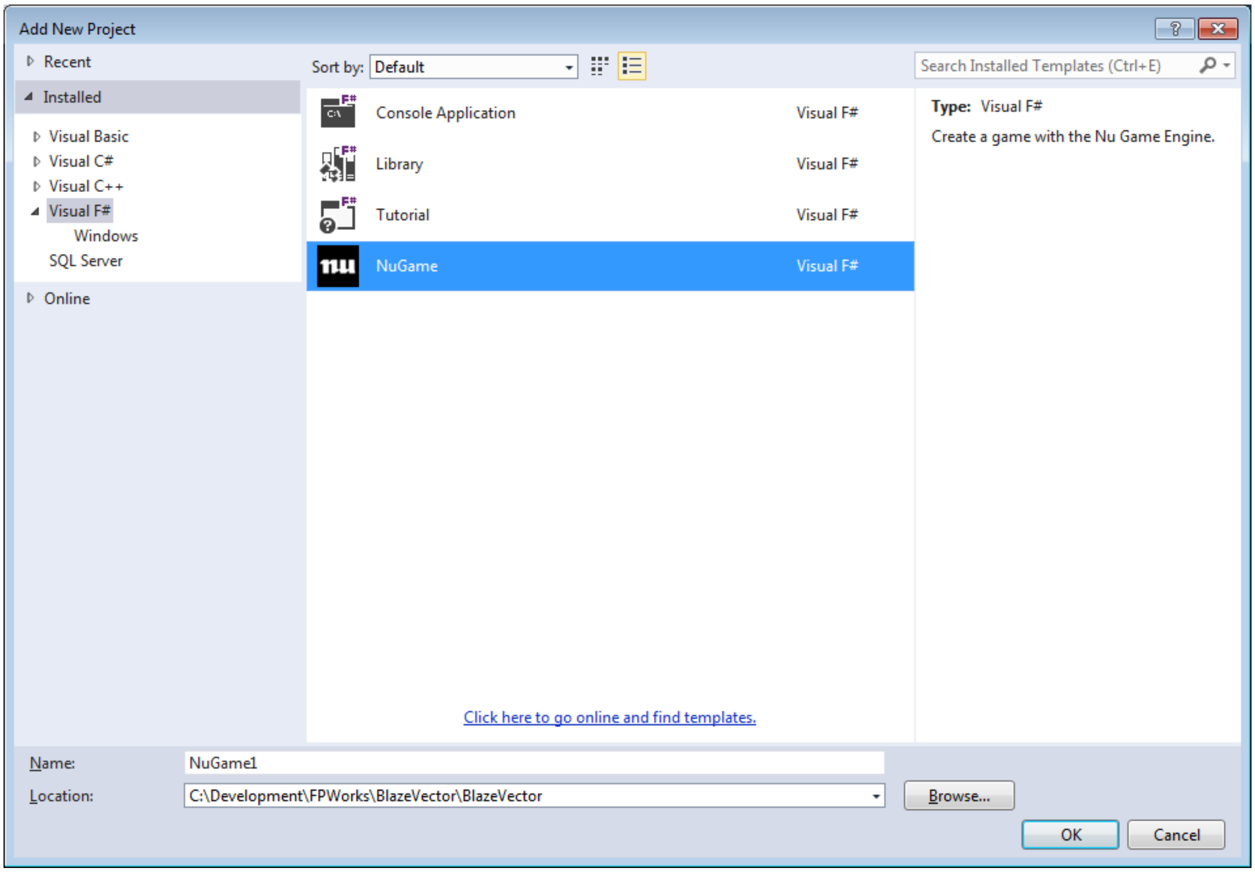
Once the solution builds successfully, ensure that the **BlazeVector** project is set as the StartUp project, and then run the game by pressing the |> Start button in Visual Studio.

Creating your own Nu game Project

Next, let's build your own game project using the Nu Game Engine.

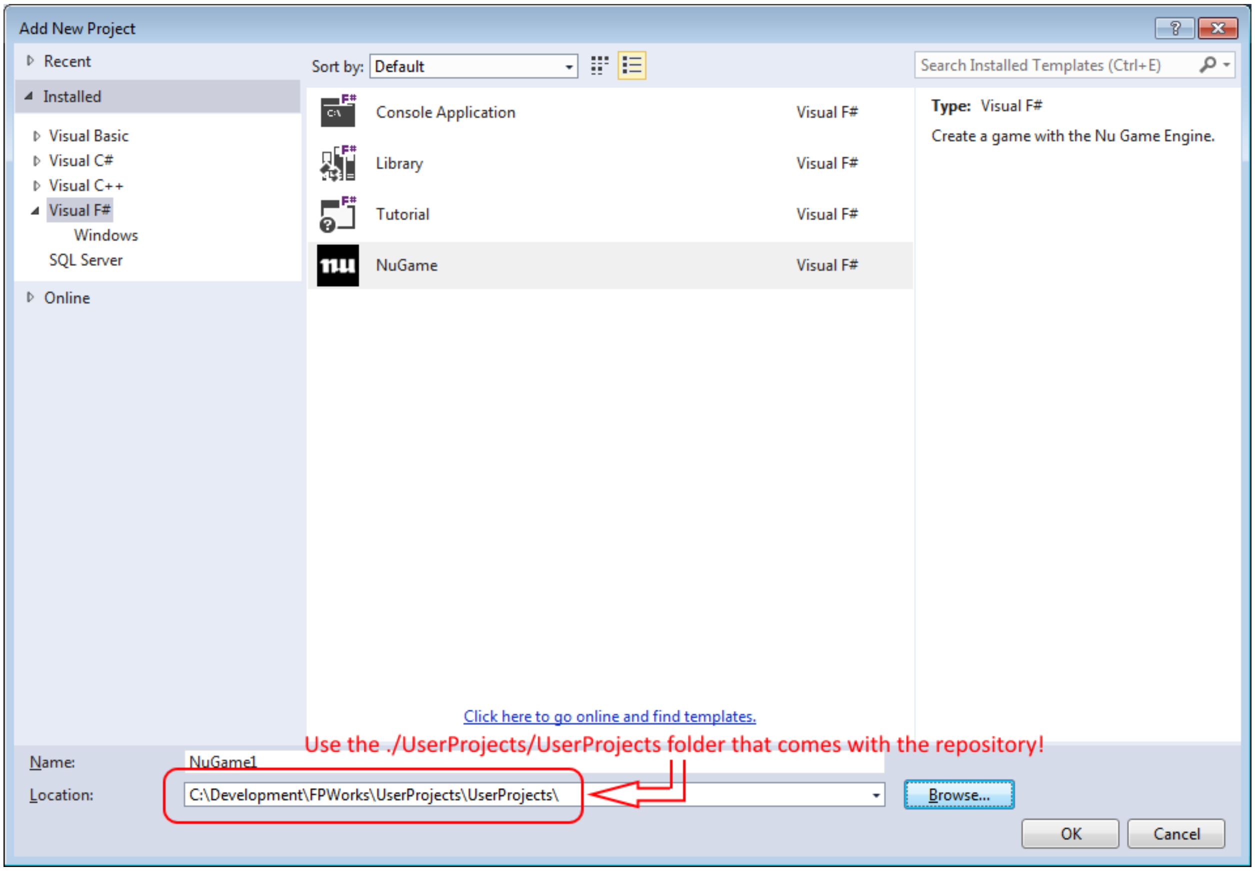
First, navigate to the ./Nu/Nu/NuTemplateExport folder and double-click the **Install.bat** file. This will install the NuGame Visual Studio project template.

Now, back in the BlazeVector solution in Visual Studio, click File -> Add -> New Project. Under the Visual F# category, select the NuGame template like so –



Next, in the Name field, enter the name of your game.

IMPORTANT: Set the Location field to the ./UserProjects/UserProjects folder like so –



If this is done incorrectly, the new project will not be able to find the Nu, NuPipe , Prime, and SDL2# dependencies needed to build it!

Finally, click **OK** to create the project. Finally try running it by setting it as the StartUp Project and then pressing the |> Start button in Visual Studio.

When the project is run from Visual Studio, you'll notice a window popping up that is filled with a nice white color. By default, Nu does nothing but clear the frame buffer with white pixels. There is no interactivity in this program, as the engine is not yet being told to do anything.

Though this is not yet an interesting program, a look at the code behind it should be enlightening.

Basic Nu Start-up Code

Here's the main code presented with comments -

```
// Nu Game Engine.
// Copyright (C) Bryan Edds, 2013-2014.

namespace NuGame1
open SDL2
open Prime
open Prime.PrimeConstants
open Nu
open Nu.NuConstants
module Program =

    // this the entry point for your application
    let [<EntryPoint>] main _ =

        // this initializes miscellaneous values required by the engine. This should always be the
        // first line in your game program.
        World.init ()

        // this specifies the manner in which the game is viewed. With this configuration, a new
        // window is created with a title of "NuGame1".
        let sdlViewConfig =
            NewWindow
                { WindowTitle = "NuGame1"
                  WindowX = SDL.SDL_WINDOWPOS_UNDEFINED
                  WindowY = SDL.SDL_WINDOWPOS_UNDEFINED
                  WindowFlags = SDL.SDL_WindowFlags.SDL_WINDOW_SHOWN }

        // this specifies the manner in which the game's rendering takes place. With this
        // configuration, rendering is hardware-accelerated and synchronized with the system's
        // vertical re-trace, making for fast and smooth rendering.
        let sdlRendererFlags =
            enum<SDL.SDL_RendererFlags>
                (int SDL.SDL_RendererFlags.SDL_RENDERER_ACCELERATED |||
                 int SDL.SDL_RendererFlags.SDL_RENDERER_PRESENTVSYNC)

        // this makes a configuration record with the specifications we set out above.
        let sdlConfig =
            { ViewConfig = sdlViewConfig
              ViewW = ResolutionX
              ViewH = ResolutionY
              RendererFlags = sdlRendererFlags
              AudioChunkSize = AudioBufferSizeDefault }

        // this is a callback that attempts to make 'the world' in a functional programming
        // sense. In a Nu game, the world is represented as a complex record type named World.
        let tryMakeWorld sdlDeps =

            // Game dispatchers specify some unique, high-level behavior and data for your game.
            // Since this particular program has no unique behavior, the vanilla base class
            // GameDispatcher is used.
            let gameDispatcher = GameDispatcher () :> obj

            // here is an attempt to make the world using SDL dependencies that will be created
            // from the invoking function using the SDL configuration that we defined above, the
            // gameDispatcher immediately above, and a value that could have been used to
            // user-defined data to the world had we needed it (we don't, so we pass unit).
            World.tryMakeEmpty sdlDeps gameDispatcher true ()

        // this is a callback that specifies your game's unique behavior when updating the world
        // every tick. The World value is the state of the world after the callback has transformed
        // the one it receives. It is here where we first clearly see Nu's purely-functional(ish)
        // design. The World type is almost entirely immutable, and thus the only way to update it
        // is by making a new copy of an existing instance. Since we need no special update
        // behavior in this program, we simply return the world as it was received.
```

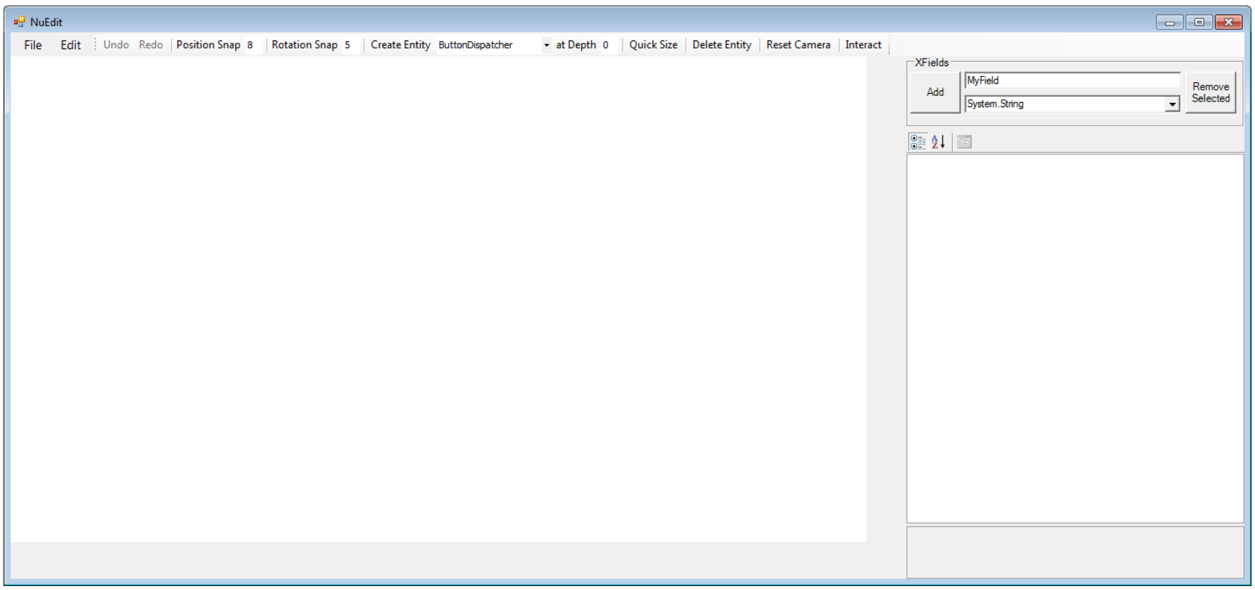
```
let updateWorld world = world

// after some configuration it is time to run Nu. We're off and running!
World.run tryMakeWorld updateWorld sdlConfig
```

Before discussing Nu's game engine design and how to customize your game, let's have a little fun messing around with Nu's real-time interactive game editor, NuEdit.

What is NuEdit?

NuEdit is Nu's game editor. Here is a screenshot of an empty editing session –

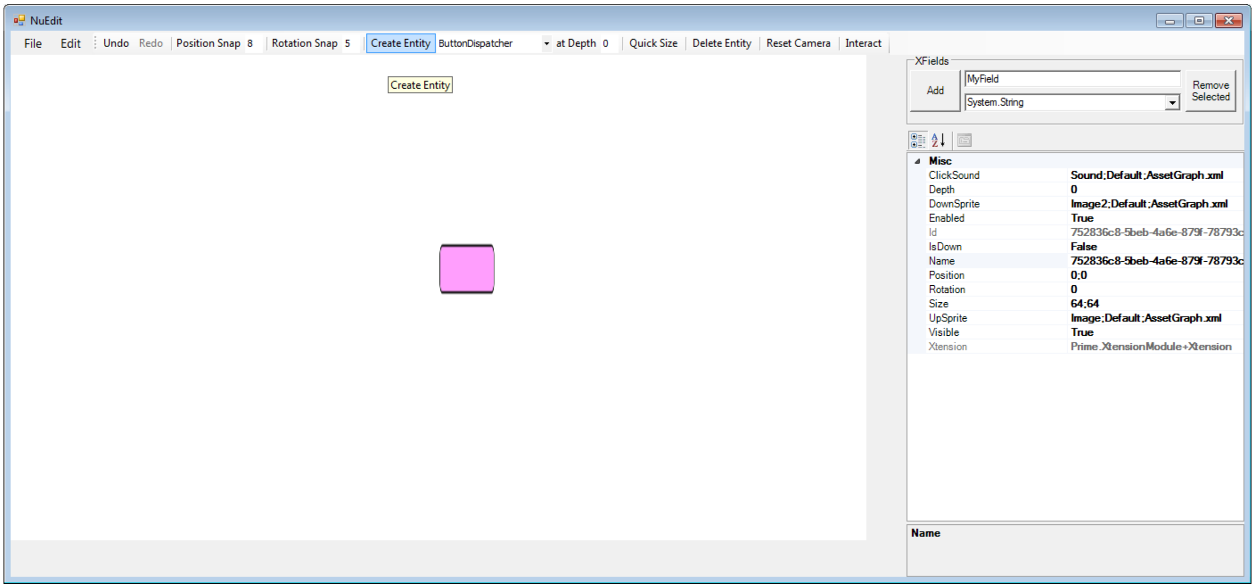


NOTE: *There may still be some stability issues with NuEdit, so save your documents early and often, and for goodness' sake use a source control system!*

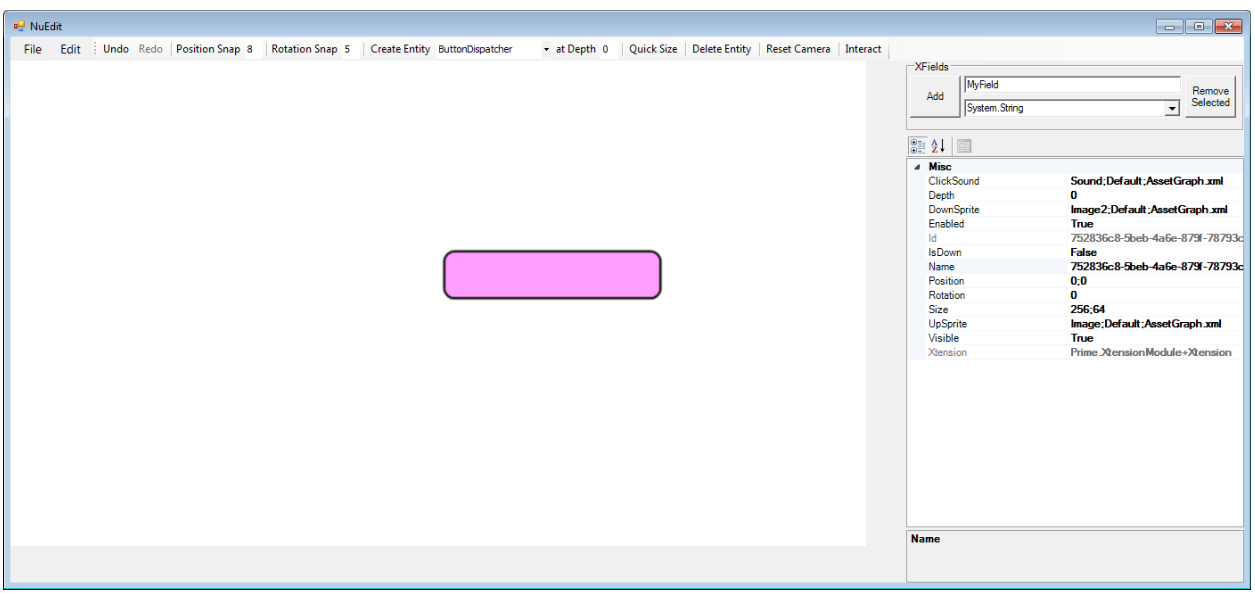
Run NuEdit by setting the NuEdit project as the StartUp Project in Visual Studio, and then running.

You'll instantly notice an Open File dialog appear from which you are instructed to "Select your game's executable file…" If you select an executable .NET file that contains concrete sub-classes of an Entity2dDispatcher, they will be made available for use in the editor.

First, we'll create a blank button by ensuring that ButtonDispatcher is selected in the combo box to the right of the Create Entity button on the main tool bar, and then pressing the Create Entity button.



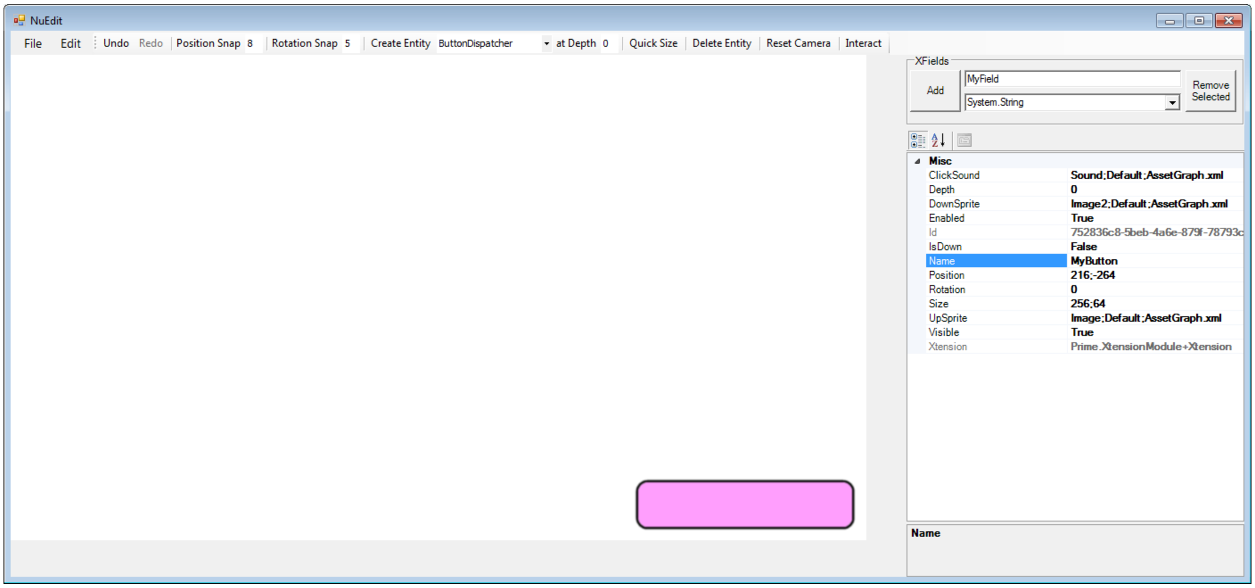
You'll notice a squished button appear in the middle of the editing panel. By default, most entities are created with a size of 64x64. Fortunately, Nu gives you an easy way to resize the entity to fit the button's image by pressing the Quick Size button. Press it now.



We have a full-sized button! Notice the property grid on the right got filled with various field names and their corresponding values. These values can be edited manually. For an entity that will be used to control the game's state (like a button), the first thing you will want to do is to give it an appropriate name. Simply double-click the Name field, delete the contents, and then enter the text "MyButton". Naming entities give you the ability to access them at runtime via that name once you have loaded the containing document in your game.

Notice also that you can click and drag on the button to move it about the screen. Once an entity is selected, you can also right-click it for more operations.

Here we've renamed the button and moved it to the bottom right of the screen –



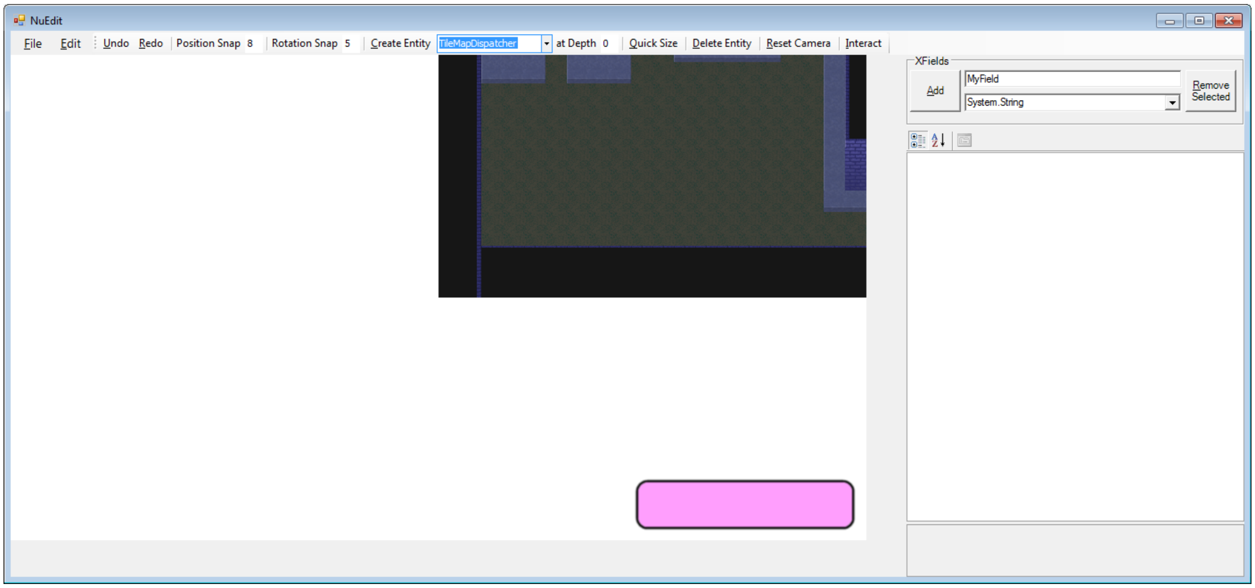
Notice you have the full power of undo and redo. Nonetheless, you should still save your documents often in case this early version of NuEdit goes bananas on you.

Let's now try putting NuEdit in interactive mode so that we can test that our button clicks as we expect. Toggle on the Interact button at the top right, then click on the button.

Once you're satisfied, toggle off the Interact button to return to editing mode.

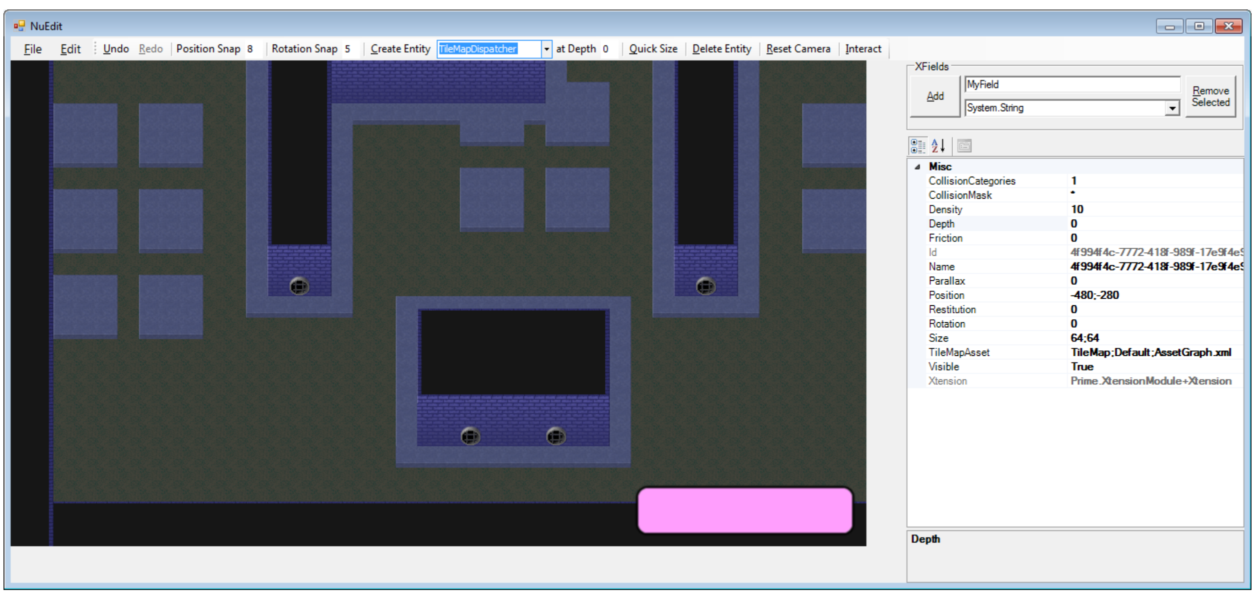
Now let's make a default tile map to play around with. BUT FIRST, we need to change the depth of our button entity so that it doesn't get covered by the new tile map! Change the value in the button's Depth field to 10.

In the drop down box to the right of the Create Entity button, select (or type) TileMapDispatcher, and then press the Create Entity button, and then click the Quick Size button. You'll get this –



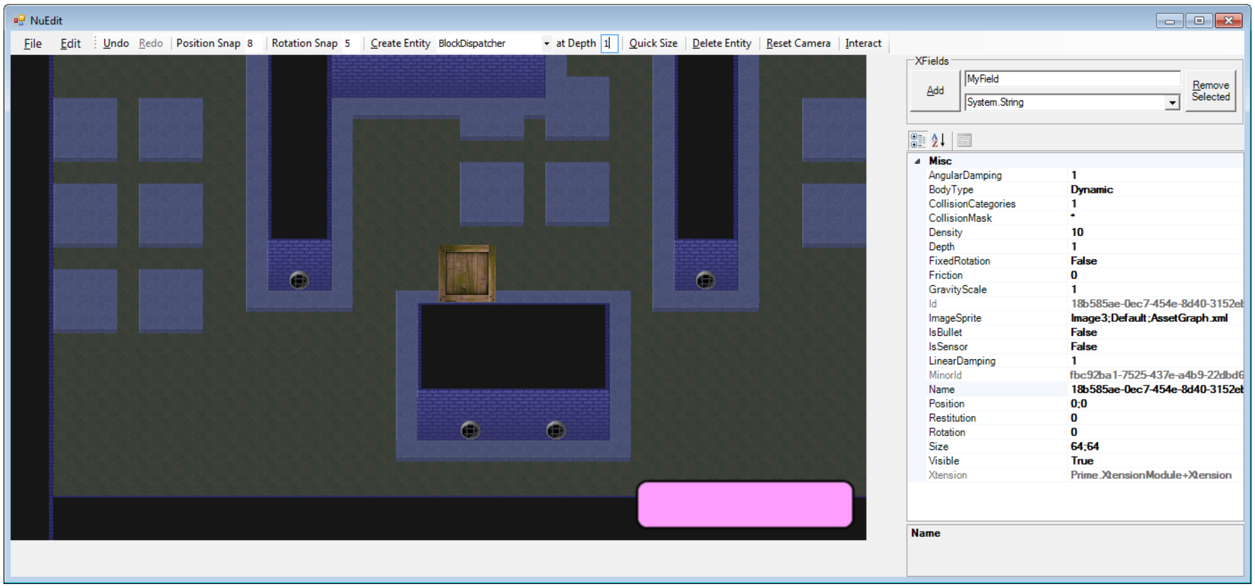
Click and drag the tile map so its bottom-left corner lines up with the top left of the editing panel.

Tile maps, by the way, are created with the free tile map editor Tiled found at http://www.mapeditor.org/. All credit to the great chap who made and maintains it!

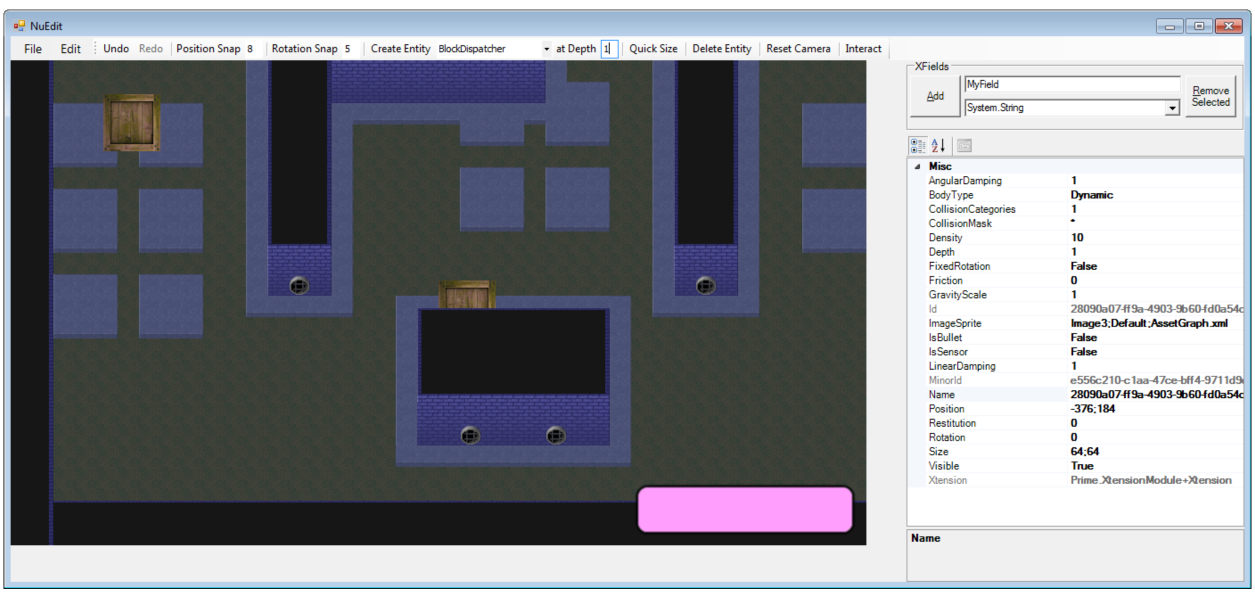


Now click and drag with the MIDDLE mouse button to change the position of the camera that is used to view the game. Check out your lovely new tile map! If your camera gets lost in space, click the Reset Camera button that is to the left of the Interact button.

Now let's create some blocks to fall down and collide with the tile map using physics. First, we must change the default depth at which new entities are created (again, so the tile map doesn't overlap them). In the at Depth text box to the left of the Quick Size button, type in a 1. In the combo box to the right of the Create Entity button, select (or type) BlockDispatcher, and then click the Create Entity button. You'll see a box created in the middle of the screen that falls directly down.



Notice that you can create blocks in other places by right-clicking at the desired location and then, in the context menu that pops up, clicking Create.



Blocks can be clicked and dragged around like other entities.

We can now save the document for loading into a game by clicking File -> Save…

Lastly, we can add custom fields (known as XFields) to each entity by selecting it on the screen and pressing the Add button in the XFields box atop the property grid. We have no use for this now, however, so we won't click anything further.

Let's watch Nu in action by returning to the sample game, BlazeVector.

BlazeVector

This is the sample game for the Nu Game Engine. In Visual Studio, set the BlazeVector project as the StartUp Project, and then run. We can look at how Nu hangs together a bit by studying BlazeVector's top level code.

First, however, we need to go over the constants that BlazeVector uses. These are defined in the BlazeConstants.fs file –

```
namespace BlazeVector
open Nu
open Nu.NuConstants
module BlazeConstants =

    // misc constants. These, and the following constants, will be explained in depth later. Just
    // scan over them for now, or look at them in the debugger on your own.
    let GuiPackageName = "Gui"
    let StagePackageName = "Stage"
    let StagePlayerName = "Player"
    let StagePlayName = "StagePlay"
    let StagePlayFileName = "Assets/BlazeVector/Groups/StagePlay.nugroup"
    let SectionName = "Section"
    let Section0FileName = "Assets/BlazeVector/Groups/Section0.nugroup"
    let Section1FileName = "Assets/BlazeVector/Groups/Section1.nugroup"
    let Section2FileName = "Assets/BlazeVector/Groups/Section2.nugroup"
    let Section3FileName = "Assets/BlazeVector/Groups/Section3.nugroup"
    let SectionFileNames = [Section0FileName; Section1FileName; Section2FileName; Section3FileName]
    let SectionCount = 32

    // asset constants
    let NuSplashSound = { SoundAssetName = "Nu"; PackageName = GuiPackageName }
    let MachinerySong = { SongAssetName = "Machinery"; PackageName = GuiPackageName }
    let DeadBlazeSong = { SongAssetName = "DeadBlaze"; PackageName = StagePackageName }
    let HitSound = { SoundAssetName = "Hit"; PackageName = StagePackageName }
    let ExplosionSound = { SoundAssetName = "Explosion"; PackageName = StagePackageName }
    let ShotSound = { SoundAssetName = "Shot"; PackageName = StagePackageName }
    let JumpSound = { SoundAssetName = "Jump"; PackageName = StagePackageName }
    let DeathSound = { SoundAssetName = "Death"; PackageName = StagePackageName }
    let EnemyBulletImage = { ImageAssetName = "EnemyBullet"; PackageName = StagePackageName }
    let PlayerBulletImage = { ImageAssetName = "PlayerBullet"; PackageName = StagePackageName }
    let EnemyImage = { ImageAssetName = "Enemy"; PackageName = StagePackageName }
    let PlayerImage = { ImageAssetName = "Player"; PackageName = StagePackageName }

    // transition constants
    let IncomingTimeSplash = 60L
    let IncomingTime = 20L
    let IdlingTime = 60L
    let OutgoingTimeSplash = 40L
    let OutgoingTime = 30L
    let StageOutgoingTime = 90L

    // splash constants
    let SplashAddress = addr "Splash"

    // title constants
    let TitleAddress = addr "Title"
    let TitleGroupAddress = addr "Title/Group"
    let SelectTitleEventName = addr "Select/Title"
    let ClickTitlePlayEventName = addr "Click/Title/Group/Play"
    let ClickTitleCreditsEventName = addr "Click/Title/Group/Credits"
    let ClickTitleExitEventName = addr "Click/Title/Group/Exit"
    let TitleGroupFileName = "Assets/BlazeVector/Groups/Title.nugroup"

    // stage constants
    let StageAddress = addr "Stage"
    let StageGroupAddress = addr "Stage/Group"
    let ClickStageBackEventName = addr "Click/Stage/Group/Back"
    let StageGroupFileName = "Assets/BlazeVector/Groups/StageGui.nugroup"
```

```
    // credits constants
    let CreditsAddress = addr "Credits"
    let CreditsGroupAddress = addr "Credits/Group"
    let ClickCreditsBackEventName = addr "Click/Credits/Group/Back"
    let CreditsGroupFileName = "Assets/BlazeVector/Groups/Credits.nugroup"
```

Nothing terribly interesting, so let's jump to Program.fs -

```
namespace BlazeVector
open System
open SDL2
open Nu
open Nu.NuConstants
open BlazeVector
module Program =

    // this the entry point for the BlazeVector application
    let [<EntryPoint>] main _ =

        // this initializes miscellaneous values required by the engine. This should always be the
        // first line in your game program.
        World.init ()

        // this specifies the manner in which the game is viewed. With this configuration, a new
        // window is created with a title of "BlazeVector".
        let sdlViewConfig =
            NewWindow
                { WindowTitle = "BlazeVector"
                  WindowX = SDL.SDL_WINDOWPOS_UNDEFINED
                  WindowY = SDL.SDL_WINDOWPOS_UNDEFINED
                  WindowFlags = SDL.SDL_WindowFlags.SDL_WINDOW_SHOWN }

        // this specifies the manner in which the game's rendering takes place. With this
        // configuration, rendering is hardware-accelerated and synchronized with the system's
        // vertical re-trace, making for fast and smooth rendering.
        let sdlRendererFlags =
            enum<SDL.SDL_RendererFlags>
                (int SDL.SDL_RendererFlags.SDL_RENDERER_ACCELERATED |||
                 int SDL.SDL_RendererFlags.SDL_RENDERER_PRESENTVSYNC)

        // this makes a configuration record with the specifications we set out above.
        let sdlConfig =
            { ViewConfig = sdlViewConfig
              ViewW = ResolutionX
              ViewH = ResolutionY
              RendererFlags = sdlRendererFlags
              AudioChunkSize = AudioBufferSizeDefault }

        // after some configuration it is time to run the game. We're off and running!
        World.run
            (fun sdlDeps -> BlazeFlow.tryMakeBlazeVectorWorld sdlDeps false ())
            (fun world -> world)
            sdlConfig
```

Well, honestly, we've seen most of this before, except the window is titled "BlazeVector" and the world creation callback is BlazeFlow.tryMakeBlazeVectorWorld instead of World.tryMakeEmptyWorld. Let's investigate into BlazeFlow.tryMakeBlazeVectorWorld to learn a little more –

```
namespace BlazeVector
open System
open Prime
open Nu
open Nu.NuConstants
open BlazeVector
open BlazeVector.BlazeConstants
module BlazeFlow =

    // this function handles playing the song "Machinery"
    let handlePlaySongMachinery _ world =
        let world = World.playSong MachinerySong 1.0f 0 world
```

```
        (Unhandled, world)

    // this function handles playing the stage
    let handlePlayStage _ world =
        let world = World.fadeOutSong DefaultTimeToFadeOutSongMs world
        let world = World.transitionScreen StageAddress world
        (Unhandled, world)

    // this function adds the BlazeVector title screen to the world
    let addTitleScreen world =

        // this adds a dissolve screen from the specified file with the given parameters
        let world = World.addDissolveScreenFromFile typeof<ScreenDispatcher>.Name TitleGroupFileName (Address.last TitleGroupAddress) IncomingTime
OutgoingTime TitleAddress world

        // this subscribes to the event that is raised when the Title screen is selected for
        // display and interaction, and handles the event by playing the song "Machinery"
        let world = World.subscribe4 SelectTitleEventName Address.empty (CustomSub handlePlaySongMachinery) world

        // subscribes to the event that is raised when the Title screen's Play button is
        // clicked, and handles the event by transitioning to the Stage screen
        let world = World.subscribe4 ClickTitlePlayEventName Address.empty (CustomSub handlePlayStage) world

        // subscribes to the event that is raised when the Title screen's Credits button is
        // clicked, and handles the event by transitioning to the Credits screen
        let world = World.subscribe4 ClickTitleCreditsEventName Address.empty (ScreenTransitionSub CreditsAddress) world

        // subscribe4s to the event that is raised when the Title screen's Exit button is clicked,
        // and handles the event by exiting the game
        World.subscribe4 ClickTitleExitEventName Address.empty ExitSub world

    // pretty much the same as above, but for the Credits screen
    let addCreditsScreen world =
        let world = World.addDissolveScreenFromFile typeof<ScreenDispatcher>.Name CreditsGroupFileName (Address.last CreditsGroupAddress) IncomingTime
OutgoingTime CreditsAddress world
        World.subscribe4 ClickCreditsBackEventName Address.empty (ScreenTransitionSub TitleAddress) world

    // and so on.
    let addStageScreen world =
        let world = World.addDissolveScreenFromFile typeof<StageScreenDispatcher>.Name StageGroupFileName (Address.last StageGroupAddress) IncomingTime
StageOutgoingTime StageAddress world
        World.subscribe4 ClickStageBackEventName Address.empty (ScreenTransitionSub TitleAddress) world

    // here we make the BlazeVector world in a callback from the World.run function.
    let tryMakeBlazeVectorWorld sdlDeps extData =

        // our custom game dispatcher here is OmniGameDispatcher
        let gameDispatcher = BlazeVectorDispatcher () :> obj

        // we use World.tryMakeEmpty to create an empty world that we will transform to create the
        // BlazeVector world
        let optWorld = World.tryMakeEmpty sdlDeps gameDispatcher GuiAndPhysicsAndGamePlay extData
        match optWorld with
        | Left _ as left -> left
        | Right world ->

            // hint to the renderer that the Gui package should be loaded up front
            let world = World.hintRenderingPackageUse GuiPackageName world

            // add our UI screens to the world
            let world = addTitleScreen world
            let world = addCreditsScreen world
            let world = addStageScreen world

            // add to the world a splash screen that automatically transitions to the Title screen
            let splashScreenImage = { ImageAssetName = "Image5"; PackageName = DefaultPackageName }
            let world = World.addSplashScreenFromData TitleAddress SplashAddress typeof<ScreenDispatcher>.Name IncomingTimeSplash IdlingTime
OutgoingTimeSplash splashScreenImage world

            // play a neat sound effect, and select the splash screen
            let world = World.playSound NuSplashSound 1.0f world
            let world = World.selectScreen SplashAddress world

            // return our world within the expected Either type, and we're off!
            Right world
```

This gives us a good idea how everything you see in the game is created and hooked together. There are far more details on the game's implementation in BlazeDispatchers.fs, but documentation on that is not yet available. Fortunately, the code is available for you to study.

As you might notice in the code shown, there is no mutation going on that is visible to the end-user. Immutability is a cornerstone of Nu's design and implementation. Remember the Undo and Redo features in NuEdit? Those are implemented simply by keeping references to past and future world values, rewinding and fast-forwarding to them as needed. This approach is a massive improvement over the complicated and fragile imperative 'Command Design Pattern' approach.

The Game Engine

You might now have a vague idea of how Nu is used and structured. Let's try to give you a clearer idea.

First and foremost, Nu was designed for *games*. This may seem an obvious statement, but it has some implications that vary it from other middleware technologies, including most game engines!

Nu comes with an appropriate game structure out of the box, allowing you to house your game's implementation inside of it. Here's the overall structure of a game as prescribed by Nu –

```
World ---> Game ---> [Screen] ---> [Group] ---> [Entity]
```

In the above diagram, X --> [Y] denotes a one-to-many relationship, and [X] --> [Y] denotes that each X has a one-to-many relationship with Y. So for example, there is only one Game in existence, but it can contain many Screens (such as a Title Screen and a Credits Screen). And for each screen, it may contain multiple Groups, each under which collections of Entities may be cohered.

Everyone should know by now that UIs are an intrinsic part of games. Rather than tacking on a UI system like other engines, Nu implements its UI components directly as entities. There is no arbitrary divide between a Block entity in the game and a Button entity.

Let's break down what each of Nu's most important types mean in detail.

World

We already know a bit about the World type. As you can see in the above diagram, it holds the Game value. It also holds all the other facilities needed to execute a game such as a rendering context, an audio context, their related message queues (more on this later), a purely-functional message system (far more appropriate to a functional game than .NET's or even F#'s mutable event systems), and other types of dependencies and configuration values. When you want something in your game to change, you start at the World and work your way inward.

Screen

Screens are precisely what they sound like – a way to implement a single 'screen' of interaction in your game. In Nu's conceptual model, a game is nothing more than a series of interactive screens to be traversed like a graph. The main game simulation occurs within a given screen, just like everything else. How screens transition from one to another is specified in code. In fact, we've already seen the code that does this in the BlazeVector.BlazeFlow.addTitleScreen function that we studied some pages above.

Group

A Group represents a logical 'grouping' of entities. NuEdit actually builds one group of entities at a time. At run-time, multiple of those groups can have their files loaded into a single screen.

Entity

And here we come down to brass tacks. Entities represent individual interactive things in your game. We've seen several already – a button, a tile map, and blocks. What differentiates a button entity from a block entity, though? Each entity picks up its unique attributes from its XDispatcher. What is a XDispatcher? Well, it's a little complicated, so we'll touch on that later!

Engine Details

Addresses

You may be wondering about the details of connecting code-driven behavior to entities created in the editor and loaded from a file at run-time. Accessing entities, including the ones loaded from a file is done with Nu's realization of 'addresses'. Each entity has an address of the form 'ScreenName/GroupName/EntityName', where the ScreenName is the name that is given to the containing Screen value, GroupName is the name given to the containing Group value, and Entity name is the name given to the Entity. Remember how we changed the Name field of the button object that we created to "MyButton" earlier in this document? That's what we're talking about, and the entity's name is just the last part of its address.

Transformations

Given all this, how do we actually make transformations to a given entity in the world?

Well, first we need to find the thing in the world that we want to transform. Then we have to transform it, and then finally place the transformed value unto a new copy of the world.

Here's some code that grabs an entity at a specific address using the getEntity function –

```
let buttonAddress = addr "TitleScreen/MainGroup/MyButton"
let button = World.getEntity buttonAddress world
```

Note that in this (and in the following code) we presume that both the Prime and OpenTK namespaces are open.

This will return an entity value at the given address. Now let's transform that button, say, by disabling it.

```
let button = Entity.setEnabled false button
```

Finally, we place the transformed value unto a new copy of the old world using the setEntity function –

```
let world = World.setEntity buttonAddress button world
```

Purely-Functional Event System

TODO

The Xtension System

In Nu, the Game, Screen, Group, and Entity types (collectively known as the simulation types) need the following capabilities that go beyond what F# provides out of the box –

1. Open extensibility without resorting to object-orientation.
2. Dynamic data augmentation / composition at run-time.
3. Dynamic behavior specification / extension at run-time.

These capabilities are provided the simulation types via the Xtension type found in Prime. We can see how Xtensions are used to provide these capabilities to the Entity type by studying it here –

```
type [<CLIMutable; StructuralEquality; NoComparison>] Entity =
    { Id : Guid
      Name : string
       Position : Vector2
      Depth : single
      Size : Vector2
      Rotation : single
      Visible : bool
      Xtension : Xtension }

    static member (?) (this : Entity, memberName) =
        fun args ->
            Xtension.(?) (this.Xtension, memberName) args

    static member (?<-) (this : Entity, memberName, value) =
        let xtension = Xtension.(?<-) (this.Xtension, memberName, value)
        { this with Xtension = xtension }

    static member dispatchesAs dispatcherTargetType entity dispatcherContainer =
        Xtension.d-ispatchesAs dispatcherTargetType entity.Xtension dispatcherContainer

    // ...rest of type definition elided for brevity...
```

In many other game engines, the Entity type would be implemented as a base class. The end-user of the engine would then define one or more specific subclasses of entities via class inheritance. In functional programming, this is problematic due the inheritance having poor support for compositionality (especially at run-time!).

However, the typical functional alternatives like Discriminated Unions (DUs) also fare poorly. DUs are closed types, and in order for the end-user to compose one’s own data and behavior (especially at run-time), open types become essential.

Compositionality, dynamism, and openness are all properties that Xtensions lend to the simulation types.

Understanding the Xtension Type

Perhaps the most efficient way to exemplify the usage of an Xtension type is by discussing its unit tests. Let's take a look a snippet from Prime's Tests.fs file –

```
let [<Fact>] canAddField () =
    let xtn = Xtension.empty
    let xtn = xtn?TestField <- 5
    let fieldValue = xtn?TestField ()
    Assert.Equal (5, fieldValue)
```

For the first test, you can see we're using the Xtension type directly rather than embedding it in another type. This is not the intended usage pattern, but it does simplify things in the context of this unit test. The test here merely demonstrates that a field called **TestField** with a value of 5 can be added to an Xtension **xtn**.

At the beginning of the test, **xtn** starts out life as an Xtension value with no fields (the 'empty' Xtension). By using the dynamic (?<-) operator as shown on the third line, **xtn** is augmented with a field named **TestField** that has a value of **5**. The next line then utilizes the dynamic (?) operator to retrieve the value of the newly added field into the **fieldValue** variable. Note the surprising presence of strong typing on the **fieldValue** variable. Let's get an explanation of why we capture such strong typing here, and where capturing the typing otherwise would require a type annotation.

By using the dynamic operators, we have a reasonable syntax with which to access members of an Entity type that are composed at run-time.

We don't always get such strong typing when using Xtensions however. Consider the following where type information isn't captured –

```
let typeInfoExample () =
    let xtn = Xtension.empty
    let xtn = xtn?TestField <- 5
    let fieldValue = xtn?TestField ()
    fieldValue
```

The type of this function will be **unit -> 'a**. This is likely not what we want since we know that the returned value is intended to be of type **int**. To address this shortcoming, a type annotation is required. There are multiple ways to achieve this, but in order to maximize clarity, I suggest putting the type annotation as near as possible to its target like so –

```
let typeInfoExample () =
    let xtn = Xtension.empty
    let xtn = xtn?TestField <- 5
    let fieldValue = xtn?TestField () : int
    fieldValue
```

An **int** annotation was added to the end of the fourth line, and the function's type became **unit -> int**. This is the level of type information we typically want and expect from F# code.

How Nu uses Xtensions in practice

Having seen the use of Xtensions in the narrow context of its unit tests, we need to understand how they're actually used in Nu.

First, note that Xtension operators / members are not accessed directly, but through each containing types forwarding functions (as seen in the above Entity type definition). Further, in order to preserve the most stringent level of typing, user code doesn't use even those forwarding functions directly, but rather type extension functions like these –

```
type Entity with

    member entity.Enabled = entity?Enabled () : bool
    static member setEnabled (value : bool) (entity : Entity) : Entity = entity?Enabled <- value
```

- which when used in practice look like this –

```
    let entity = Entity.setEnabled (not entity.Enabled)
```

This is to allow user code to use the most stringent level of typing possible even though such members are in actuality dynamic! The same approach is used for Xtension dispatch methods as well (we'll go over in the next section) –

```
type Entity with

    static member init (entity : Entity) (dispatcherContainer : IXDispatcherContainer) : Entity = entity?Init (entity, dispatcherContainer)
```

- with usage like so -

```
let entity = Entity.init entity dispatcherContainer
```

Static typing with dynamic values – nearly the best of both worlds!

XDispatchers and Dispatch Methods

XDispatchers (or more casually, *dispatchers*), essentially, allow you to specify the behavior of a type that has been augmented with Xtension capabilities.

To start understanding how this works, let's take a look at the Xtension type (or at least the portion we're interested in), as well as its related types –

```
type [<StructuralEquality; NoComparison>] Xtension =
    { XFields : XFields
      OptXDispatcherName : string option
      CanDefault : bool
      Sealed : bool }

and XDispatchers =
    Map<string, obj>

and IXDispatcherContainer =
```

```
        interface
            abstract GetDispatchers : unit -> XDispatchers
            end
```

The first field, XFields, is where the dynamic field values of the Xtension types are stored. Here, the XFields type is just an alias for the F# Map<string, obj> type.

The second field, OptXDispatcherName, specifies the name of an XDispatcher type from which an Xtension gets its specialized behavior, if any.

The XDispatchers type is also an alias for the F# Map<string, obj> type, but with a different purpose than the XFields type. The XDispatchers type represents an association between an XDispatcherName and an instance of the XDispatcher that defines the behavior of the associated Xtensions.

The IXDispatcherContainer type represents a container that can provide XDispatcher references via their associated names. The definition of the type that provides these XDispatchers may be specialized by the user according to the needs of the XDispatcher. In the Nu Game Engine, the World type implements this interface while also providing a wider interface needed by EntityDispatcher and its sub-types (which are found here - https://github.com/bryanedds/FPWorks/blob/master/Nu/Nu/Nu/Dispatchers.fs).

Let's put all this in context by eyeing a unit test and some related test types that leverage this functionality –

```
    type TestDispatcher () =
        member dispatcher.Init (xtn : Xtension, _ : IXDispatcherContainer) =
            xtn?InittedField <- 5
        member dispatcher.Test (xtn : Xtension, _ : IXDispatcherContainer) =
            xtn?InittedField () * 5

    type TestDispatcherContainer () =
        let testDispatcher = (TestDispatcher ()) :> obj
        let testDispatchers = Map.singleton typeof<TestDispatcher>.Name testDispatcher
        interface IXDispatcherContainer with
            member this.GetDispatchers () = testDispatchers

    let tdc = TestDispatcherContainer ()

    let [<Fact>] dispatchingWorks () =
        let xtn = { Xtension.empty with OptXDispatcherName = Some typeof<TestDispatcher>.Name }
        let xtn = xtn?Init (xtn, tdc) : Xtension
        let dispatchResult = xtn?Test (xtn, tdc)
        Assert.Equal (dispatchResult, 25)
```

The TestDispatcher type is a simple class that demonstrates the handling of dispatches sent to an Xtension value. Notice that the IXDispatcherContainer is taken as the last argument. Due to conventions imposed by the definition of the Xtension type, **the placement scheme of this argument for all dispatches is mandatory.** The IXDispatcherContainer (or a sub-type thereof) must always be the last parameter!

Below the TestDispatcher definition, we see that the TestDispatcherContainer is defined to do little more than provide a TestDispatcher instance via the IXDispatcherContainer interface.

Immediately below that, we see a global instance of the TestDispatcherContainer **tdc** that we can reference in our unit tests.

Having looked at those definitions, let's now break down the dispatchingWorks unit test above –

```
let xtn = { Xtension.empty with OptXDispatcherName = Some typeof<TestDispatcher>.Name }
```

xtn is assigned an empty Xtension with its OptXDispatcherName set to that of the TestDispatcher's name. Because of this, dispatches sent to the Xtension will be handled by the TestDispatcher type.

```
let xtn = xtn?Init tdc : Xtension
```

Here we see the first invocation of a dynamic dispatch. Just like the dynamic field setter, the (?) dynamic lookup operator is used. Notice that in this context we need return a type annotation (the type of which is obvious as an initialization function is always sure to return a copy of the target).

Now we can call the dispatch named Dispatch and check its return value in the unit test -

```
let dispatchResult = xtn?Test tdc
Assert.Equal (dispatchResult, 25)
```

The result of Test dispatch is 25 because InittedField was defined to be 5 and because the Test dispatch was defined as –

```
member dispatcher.Test (xtn : Xtension, _ : IXDispatcherContainer) =
    xtn?InittedField () * 5
```

So 5 * 5 results in 25.

Here we see how Xtension dispatching gives us the power of dynamic dispatching without mucking up our domain model with classes (and the mutability and confusion they drag in with them). Additionally, it gives us the power to specify the dispatcher of an Xtension at run-time by altering its OptXDispatcherName field.

Of course, as pointed out above, dispatch methods are not invoked directly from the Xtension type in Nu, and user code will use forwarding functions instead of the dynamic operators directly. And lastly, Nu's World type becomes the last argument of dispatch methods because it implements the IXDispatcherContainer interface.

Facets

There are even more capabilities provided by Xtensions. Using Facets, users can use functions themselves to compose new behaviors for the simulation types. A facet is a module whose functions collectively define a single new behavior for a simulation type. Let's take a look at the definition and use of one of Nu's facets now –

```
[<AutoOpen>]
module SimpleSpriteFacetModule =

    type Entity with

        member entity.SpriteImage with get () = entity?SpriteImage () : Image
        static member setSpriteImage (value : Image) (entity : Entity) : Entity = entity? SpriteImage <- value

[<RequireQualifiedAccess>]
module SimpleSpriteFacet =

    let init (entity : Entity) (_ : IXDispatcherContainer) =
        Entity.setImage { ImageAssetName = "Image3"; PackageName = DefaultPackageName } entity

    let getRenderDescriptors entity viewType world =
        if entity.Visible && Camera.inView3 entity.Position entity.Size world.Camera then
            [LayerableDescriptor
                { Depth = entity.Depth
                  LayeredDescriptor =
                    SpriteDescriptor
                        { Position = entity.Position
```

```
                    Size = entity.Size
                    Rotation = entity.Rotation
                    ViewType = viewType
                    OptInset = None
                    Image = entity.Image
                    Color = Vector4.One }}]
        else []

    let getQuickSize (entity : Entity) world =
        let image = entity.Image
        match Metadata.tryGetTextureSizeAsVector2 image.ImageAssetName image.PackageName world.AssetMetadataMap with
        | None -> DefaultEntitySize
        | Some size -> size
```

This SimpleSpriteFacet is used to define simple sprite rendering behavior for an Entity, and may be used to define a new dispatcher like so –

```
[<AutoOpen>]
module SimpleSpriteDispatcherModule =

    type [<Sealed>] SimpleSpriteDispatcher () =
        inherit Entity2dDispatcher ()

        override dispatcher.Init (entity, dispatcherContainer) =
            let entity = base.Init (entity, dispatcherContainer)
            SimpleSpriteFacet.init entity dispatcherContainer

        override dispatcher.GetRenderDescriptors (entity, world) =
            SimpleSpriteFacet.getRenderDescriptors entity Relative world

        override dispatcher.GetQuickSize (entity, world) =
            SimpleSpriteFacet.getQuickSize entity world
```

Similar to the SimpleSpriteFacet, there is also a SimpleBodyFacet. However, instead of defining a sprite-displaying behavior, the SimpleBodyFacet defines simple physics behavior for an entity.

```
[<AutoOpen>]
module SimpleBodyFacetModule =

    type Entity with

        member entity.MinorId with get () = entity?MinorId () : Guid
        static member setMinorId (value : Guid) (entity : Entity) : Entity = entity?MinorId <- value
        member entity.BodyType with get () = entity?BodyType () : BodyType
        static member setBodyType (value : BodyType) (entity : Entity) : Entity = entity?BodyType <- value
        member entity.Density with get () = entity?Density () : single
        static member setDensity (value : single) (entity : Entity) : Entity = entity?Density <- value
        member entity.Friction with get () = entity?Friction () : single
        static member setFriction (value : single) (entity : Entity) : Entity = entity?Friction <- value
        member entity.Restitution with get () = entity?Restitution () : single
        static member setRestitution (value : single) (entity : Entity) : Entity = entity?Restitution <- value
        member entity.FixedRotation with get () = entity?FixedRotation () : bool
        static member setFixedRotation (value : bool) (entity : Entity) : Entity = entity?FixedRotation <- value
        member entity.LinearDamping with get () = entity?LinearDamping () : single
        static member setLinearDamping (value : single) (entity : Entity) : Entity = entity?LinearDamping <- value
        member entity.AngularDamping with get () = entity?AngularDamping () : single
        static member setAngularDamping (value : single) (entity : Entity) : Entity = entity?AngularDamping <- value
        member entity.GravityScale with get () = entity?GravityScale () : single
        static member setGravityScale (value : single) (entity : Entity) : Entity = entity?GravityScale <- value
        member entity.CollisionCategories with get () = entity?CollisionCategories () : string
        static member setCollisionCategories (value : string) (entity : Entity) : Entity = entity?CollisionCategories <- value
        member entity.CollisionMask with get () = entity?CollisionMask () : string
        static member setCollisionMask (value : string) (entity : Entity) : Entity = entity?CollisionMask <- value
        member entity.IsBullet with get () = entity?IsBullet () : bool
        static member setIsBullet (value : bool) (entity : Entity) : Entity = entity?IsBullet <- value
        member entity.IsSensor with get () = entity?IsSensor () : bool
        static member setIsSensor (value : bool) (entity : Entity) : Entity = entity?IsSensor <- value

        static member getPhysicsId (entity : Entity) = PhysicsId (entity.Id, entity.MinorId)

[<RequireQualifiedAccess>]
module SimpleBodyFacet =

    let init (entity : Entity) (_ : IXDispatcherContainer) =
```

```
        entity |>
            Entity.setMinorId (NuCore.makeId ()) |>
            Entity.setBodyType BodyType.Dynamic |>
            Entity.setDensity NormalDensity |>
            Entity.setFriction 0.0f |>
            Entity.setRestitution 0.0f |>
            Entity.setFixedRotation false |>
            Entity.setLinearDamping 1.0f |>
            Entity.setAngularDamping 1.0f |>
            Entity.setGravityScale 1.0f |>
            Entity.setCollisionCategories "1" |>
            Entity.setCollisionMask "*" |>
            Entity.setIsBullet false |>
            Entity.setIsSensor false

    let registerPhysics getBodyShape address world =
        let entity = World.getEntity address world
        let bodyProperties =
            { Shape = getBodyShape entity world
              BodyType = entity.BodyType
              Density = entity.Density
              Friction = entity.Friction
              Restitution = entity.Restitution
              FixedRotation = entity.FixedRotation
              LinearDamping = entity.LinearDamping
              AngularDamping = entity.AngularDamping
              GravityScale = entity.GravityScale
              CollisionCategories = Physics.toCollisionCategories entity.CollisionCategories
              CollisionMask = Physics.toCollisionCategories entity.CollisionMask
              IsBullet = entity.IsBullet
              IsSensor = entity.IsSensor }
        let physicsId = Entity.getPhysicsId entity
        let position = entity.Position + entity.Size * 0.5f
        let rotation = entity.Rotation
        World.createBody address physicsId position rotation bodyProperties world

    let unregisterPhysics address world =
        let entity = World.getEntity address world
        World.destroyBody (Entity.getPhysicsId entity) world

    let propagatePhysics getBodyShape address world =
        let world = unregisterPhysics address world
        registerPhysics getBodyShape address world

    let handleBodyTransformMessage address (message : BodyTransformMessage) world =
        let entity = World.getEntity address world
        let entity =
            entity |>
                Entity.setPosition (message.Position - entity.Size * 0.5f) |> // TODO: see if this center-offsetting can be
encapsulated within the Physics module!
                Entity.setRotation message.Rotation
        World.setEntity message.EntityAddress entity world
```

Complex behavior for a single simulation type can be defined by composing multiple facets in an XDispatcher definition. Here's an entity dispatcher that combines the SimpleSpriteFacet and SimpleBodyFacet facets –

```
[<AutoOpen>]
module SimpleBodySpriteDispatcherModule =

    type [<Sealed>] SimpleBodySpriteDispatcher () =
        inherit Entity2dDispatcher ()

        let getBodyShape (entity : Entity) _ =
            BoxShape { Extent = entity.Size * 0.5f; Center = Vector2.Zero }

        override dispatcher.Init (entity, dispatcherContainer) =
            let entity = base.Init (entity, dispatcherContainer)
            let entity = SimpleSpriteFacet.init entity dispatcherContainer
            SimpleBodyFacet.init entity dispatcherContainer

        override dispatcher.Register (address, world) =
            SimpleBodyFacet.registerPhysics getBodyShape address world

        override dispatcher.Unregister (address, world) =
```

```
        SimpleBodyFacet.unregisterPhysics address world

    override dispatcher.PropagatePhysics (address, world) =
        SimpleBodyFacet.propagatePhysics getBodyShape address world

    override dispatcher.HandleBodyTransformMessage (address, message, world) =
        SimpleBodyFacet.handleBodyTransformMessage address message world

    override dispatcher.GetRenderDescriptors (entity, world) =
        SimpleSpriteFacet.getRenderDescriptors entity Relative world

    override dispatcher.GetQuickSize (entity, world) =
        SimpleSpriteFacet.getQuickSize entity world
```

By creating new dispatchers by composing facets, arbitrarily complex Entities, Groups, and Screens can be created with relative ease.

Assets and the AssetGraph

TODO.

Serialization and Overlays

TODO.

Subsystems and Message Queues

TODO.