Nu Game Engine (pre-pre-preview)

Bryan Edds, 2014

Table of Contents

What's It All About?

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

Let me explain each of those terms –

Very Basic

Nu is very young, so it has just about no frills. Is there a particle or special effects system? Not yet, I'm afraid. Is there a sprite 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 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 Nu code, feel absolutely safe, if not 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 currently does not have a binary distribution. Instead it has a github repository at https://github.com/bryanedds/OmniBlade. I'm going to assume you know (or can quickly figure out) how to pull down the git repository on your own. Please take note of the license when pulling down the repository.

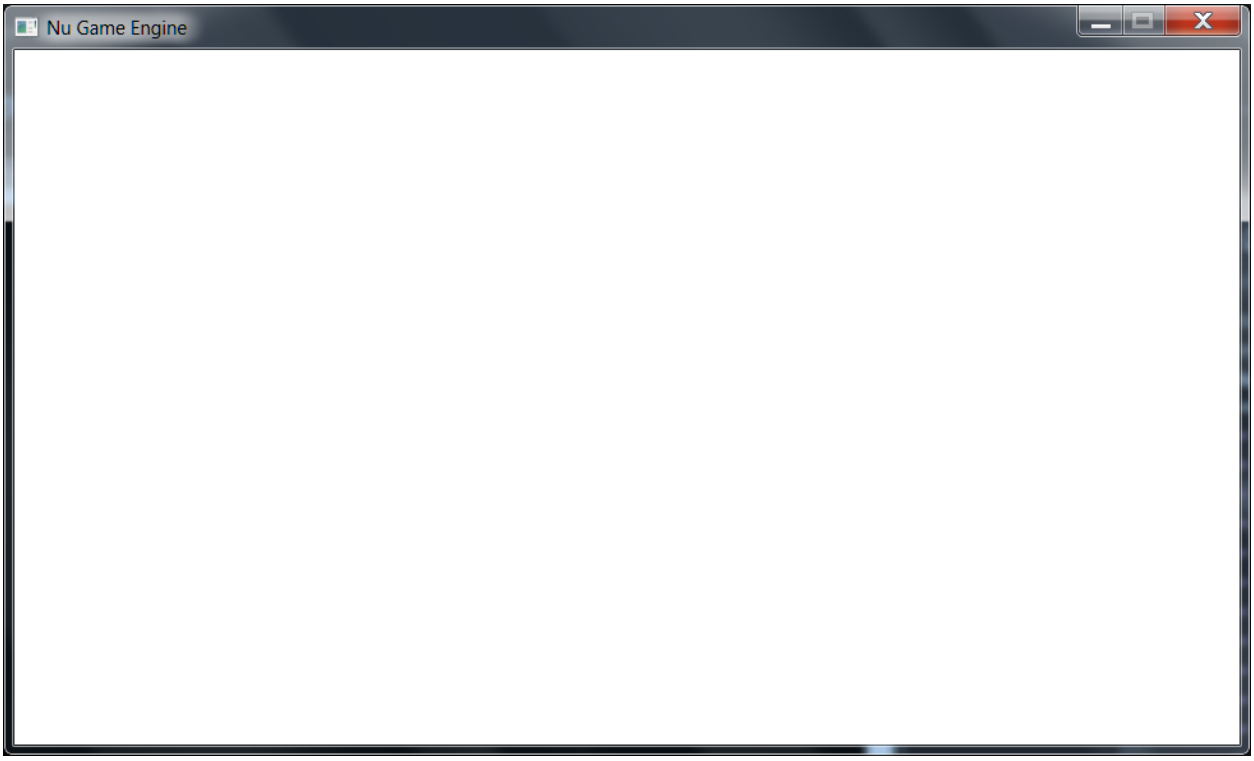
The first thing you might notice about the repository is that it contains more than just the Nu Game Engine. It also includes the source for the Aml programming language, the Prime F# code library, and some other loosely related stuff. Both Aml and Prime are required to build the Nu solution, and the rest of the stuff is safely ignored, so feel free to pull it all down.

To open the Nu solution, first make sure to have Visual Studio 2012 (or maybe 2013 – not tested!) installed. Then navigate to the ./Nu/Nu folder and open the Nu.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 you have built the solution, try running the standalone engine by setting the Nu project as the StartUp Project, and then running.

Running the Nu Project (Nu.exe)

When the app 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 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 -

```
namespace Nu
open SDL2
open Nu
module Program =

    let [<EntryPoint>] main _ =

        // this initializes all the .Net TypeConverters that Nu uses for serialization. This should
        // always be the first line in your Nu program.
        World.initTypeConverters ()

        // this specifies the manner in which Nu is viewed. With this configuration, a new window
        // is created with a title of "Nu Game Engine" and is placed at (32, 32) pixels from the
        // top left of the screen.
        let sdlViewConfig =
            NewWindow
                { WindowTitle = "Nu Game Engine"
                  WindowX = 32
                  WindowY = 32
```

```
          WindowFlags = SDL.SDL_WindowFlags.SDL_WINDOW_SHOWN }

// this specifies the manner in which Nu's rendering takes place. With this configuration,
// rendering in Nu is hardware-accelerated and synchronized with the system's vertical
// trace, making for fast and smooth rendering.
let sdlRenderFlags =
    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 =
    Sdl.makeSdlConfig
        sdlViewConfig
        Voords.VirtualResolutionX
        Voords.VirtualResolutionY
        sdlRenderFlags
        1024

// this is a callback that attempts to create 'the world' in a functional programming
// sense. In Nu, the world is represented as a complex record type name World.
let tryCreateWorld 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 create Nu's world using SDL dependencies that will be created
    // from the invoking function using the SDL configuration 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.tryCreateEmptyWorld sdlDeps gameDispatcher ()

// this is a callback that specifies your game's unique behavior when updating the world
// every tick. Its return type is a (bool * World). The bool value is whether the program
// should continue (true), or exit (false). 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 behavoir in this program, we simply return the world as it was
// received.
let updateWorld world =
    (true, world)

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

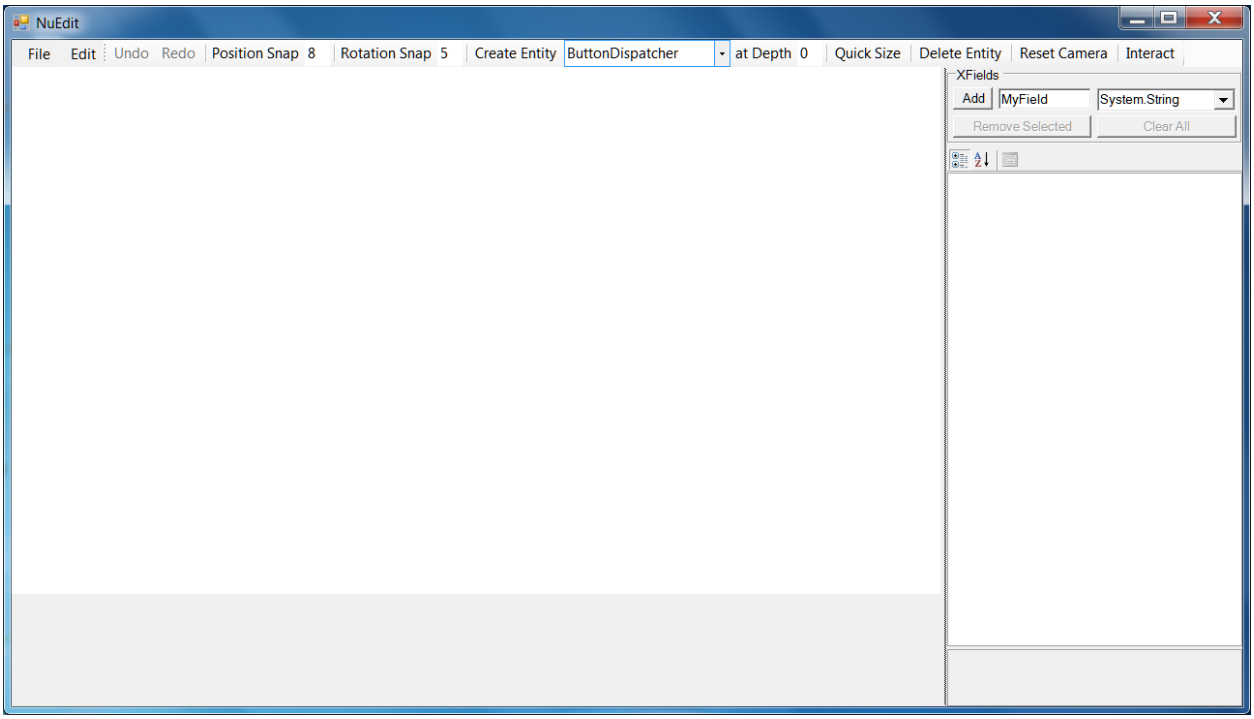
Hopefully that was somewhat enlightening. You can find this code from Visual Studio in the Program.fs file of the Nu project in the Nu.sln solution.

When creating a new Nu game project, you can copy and modify this file into your project to use as a template for your program. Creating a new Nu game project is mostly just creating a new F# program project, setting up the references, and using said code as a template.

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

What is NuEdit?

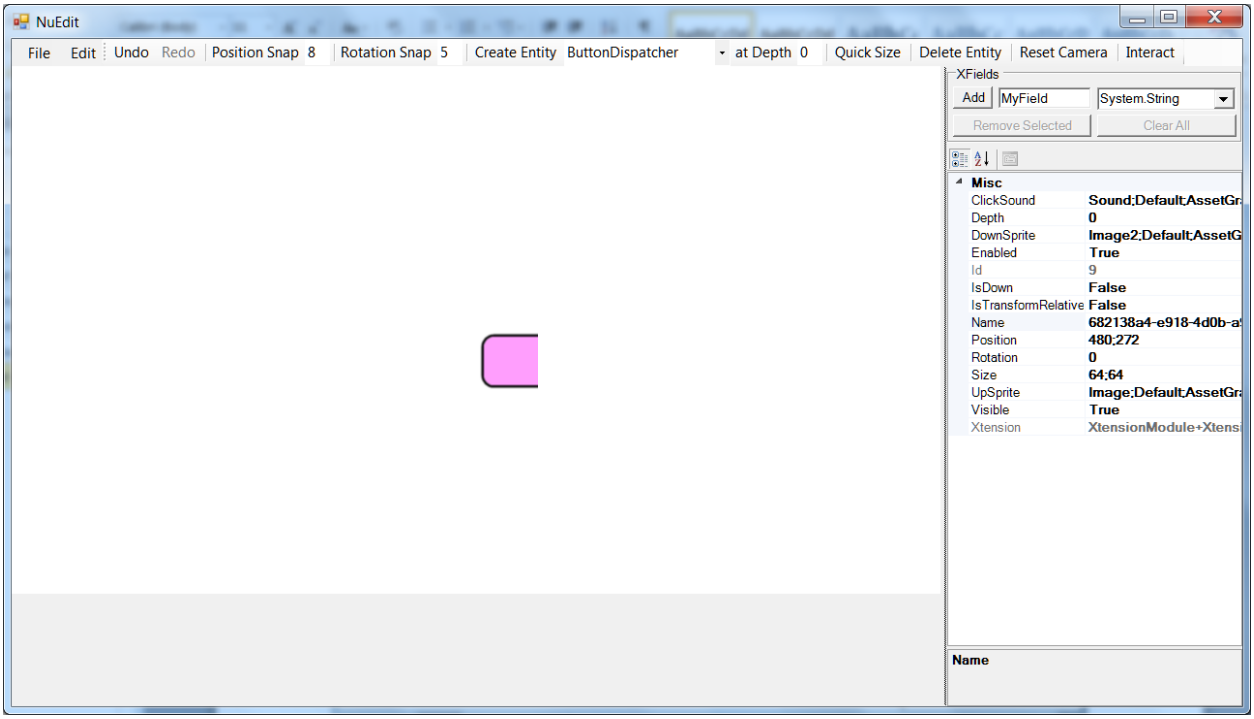
NuEdit is Nu's fairly usable game editor. Here is a screenshot of an empty editing session –



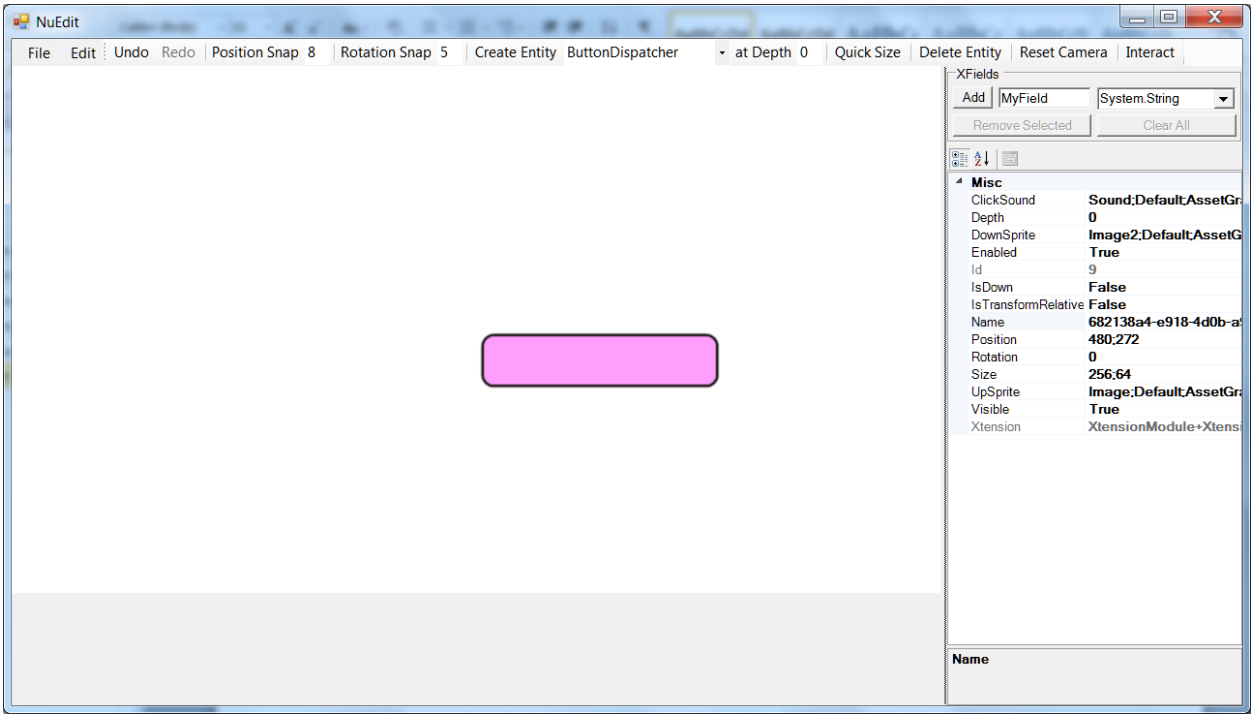
NOTE: *There may still be some stability issues with NuEdit, so save your documents early and often, and for goodness' sake use a source control system!*

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

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 portion of a button appear in the middle of the editing panel. By default, most entities are created as a 64x64 sprite. Fortunately, Nu gives you an easy way to resize the sprite to fit the button's image by pressing the Quick Size button. Press it now.

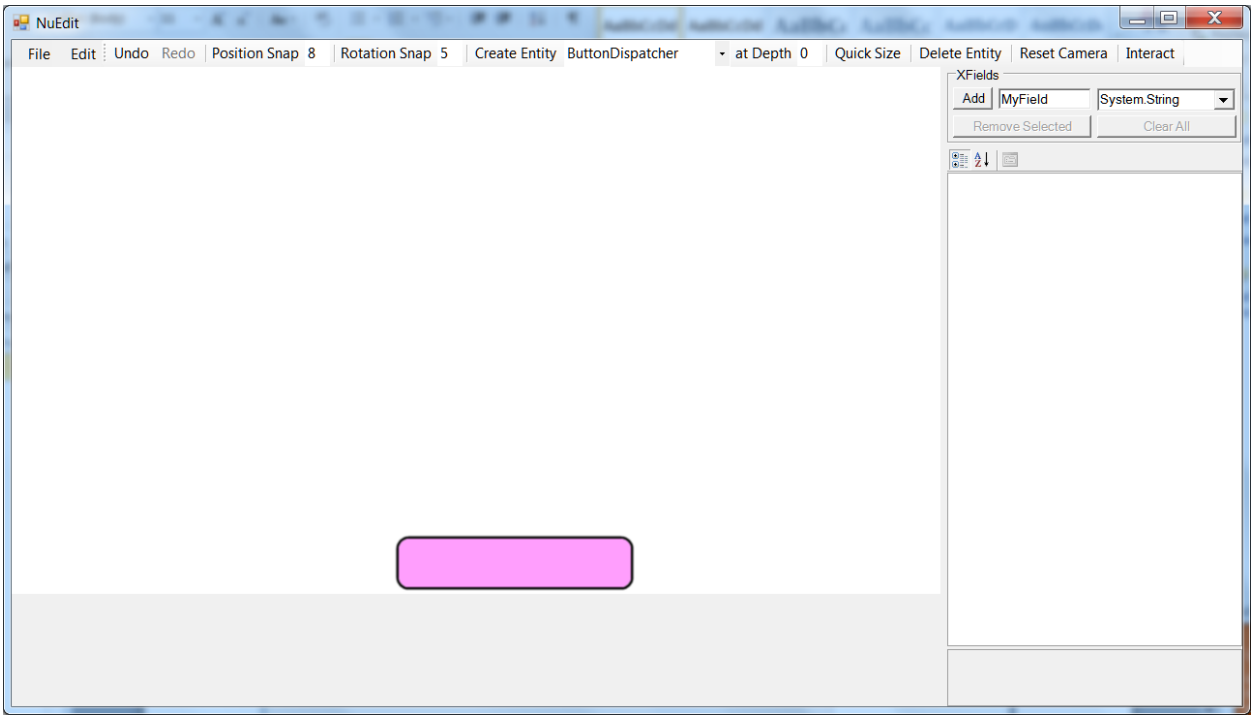


We have a full button! Notice the property grid on the right got filled with various field names and their corresponding values. These values can be edited manually. For an entity that will be used to control the

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 I've renamed the button and moved it to the bottom center of the screen –



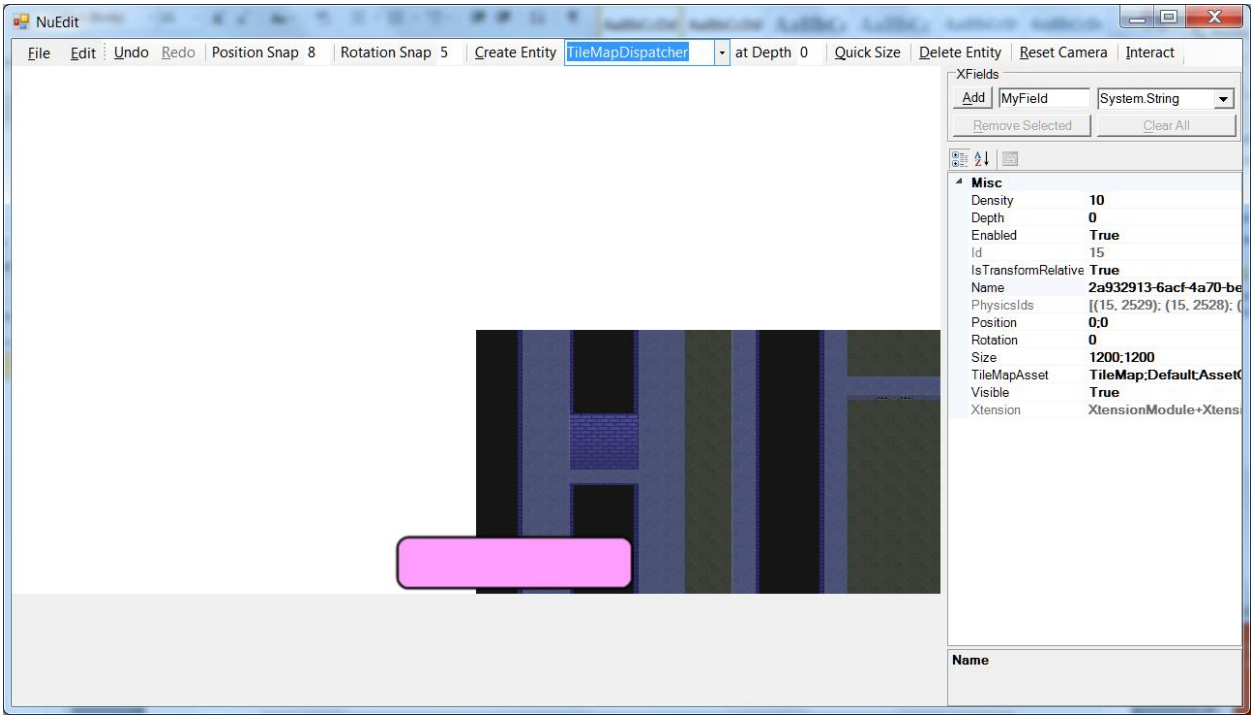
Notice you have the full power of undo and redo. Nonetheless, you should still save your documents often in case this VERY early version of NuEdit goes bananas on you.

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.

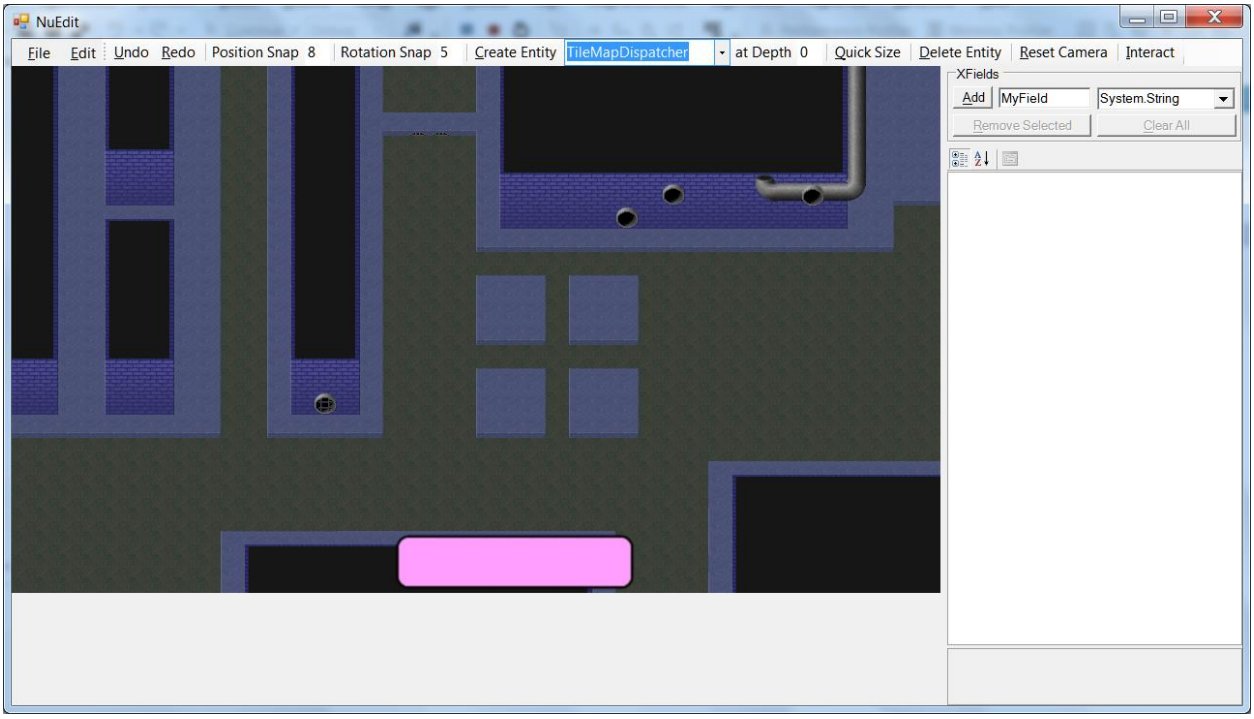
Now let's make a default tile map to play around with. BUT FIRST, we need to change the depth of our button entity so that it doesn't get covered by the new tile map! Change the value in the button's Depth field to 10.

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 –



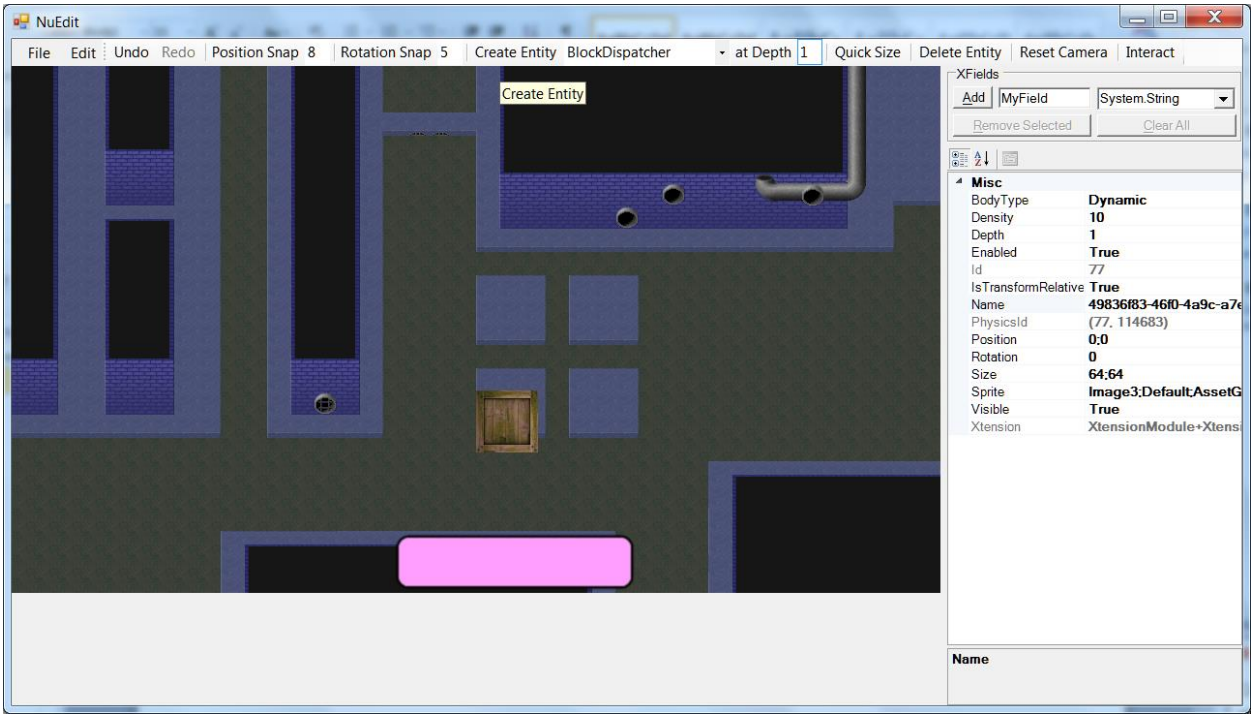
Click and drag the tile map so its top-left corner lines up with the top left of the editing panel.

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!

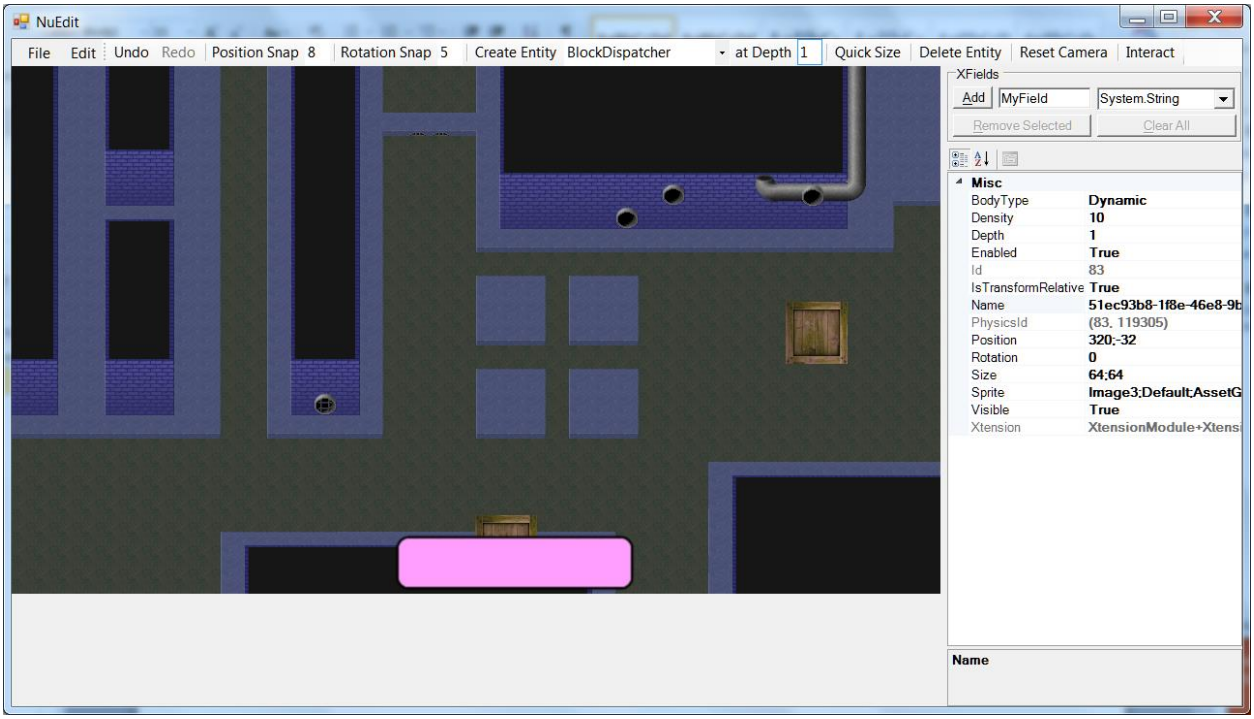


Now click and hold down the MIDDLE mouse button to change the position of the camera that is used to view the game. Check out your lovely new tile map! If your camera gets lost in space, click the Reset Camera that is to the immediate left of the Interact button.

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 with the current WIP game called OmniBlade.

OmniBlade

This is a game that's extremely early that I've just started working on. As such, it will be an interesting if shallow demo of how Nu is controlled.

In Visual Studio, set the OmniBlade project as the StartUp Project, and then run. Currently there's only a splash screen, a skeleton UI and a map to wander the circular avatar around in when going into a New Game.

Though this demo is trite and uninteresting, we can look at how Nu hangs together a bit by studying OmniBlade's main code below –

```
namespace OmniBlade
open System
open SDL2
open OpenTK
open TiledSharp
open Nu
open OmniBlade
module Program =

    let [<EntryPoint>] main _ =

        World.initTypeConverters ()

        let sdlViewConfig =
            NewWindow
                { WindowTitle = "OmniBlade"
                  WindowX = 32
                  WindowY = 32
                  WindowFlags = SDL.SDL_WindowFlags.SDL_WINDOW_SHOWN }

        let sdlRenderFlags =
            enum<SDL.SDL_RendererFlags>
                (int SDL.SDL_RendererFlags.SDL_RENDERER_ACCELERATED |||
                 int SDL.SDL_RendererFlags.SDL_RENDERER_PRESENTVSYNC)

        let sdlConfig =
            Sdl.makeSdlConfig
                sdlViewConfig
                Voords.VirtualResolutionX
                Voords.VirtualResolutionY
                sdlRenderFlags
                1024

        World.run
            (fun sdlDeps -> OmniFlow.tryCreateOmniBladeWorld sdlDeps ())
            (fun world -> (true, world))
            sdlConfig
```

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

```
namespace OmniBlade
open System
open SDL2
open OpenTK
open TiledSharp
open Nu
module OmniFlow =

    // transition 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 IncomingTimeSplash = 60
    let IncomingTime = 20
    let IdlingTime = 60
    let OutgoingTimeSplash = 40
    let OutgoingTime = 20

    // splash constants
```

```
let SplashAddress = NuCore.addr "Splash"

// title constants
let TitleAddress = NuCore.addr "Title"
let TitleGroupName = Lun.make "Group"
let TitleGroupAddress = TitleAddress @ [TitleGroupName]
let TitleGroupFileName = "Assets/OmniBlade/Groups/Title.nugroup"
let ClickTitleNewGameAddress = NuCore.straddrstr "Click" TitleGroupAddress "NewGame"
let ClickTitleLoadGameAddress = NuCore.straddrstr "Click" TitleGroupAddress "LoadGame"
let ClickTitleCreditsAddress = NuCore.straddrstr "Click" TitleGroupAddress "Credits"
let ClickTitleExitAddress = NuCore.straddrstr "Click" TitleGroupAddress "Exit"

// load game constants
let LoadGameAddress = NuCore.addr "LoadGame"
let LoadGameGroupName = Lun.make "Group"
let LoadGameGroupAddress = LoadGameAddress @ [LoadGameGroupName]
let LoadGameGroupFileName = "Assets/OmniBlade/Groups/LoadGame.nugroup"
let ClickLoadGameBackAddress = NuCore.straddrstr "Click" LoadGameGroupAddress "Back"

// credits constants
let CreditsAddress = NuCore.addr "Credits"
let CreditsGroupName = Lun.make "Group"
let CreditsGroupAddress = CreditsAddress @ [CreditsGroupName]
let CreditsGroupFileName = "Assets/OmniBlade/Groups/Credits.nugroup"
let ClickCreditsBackAddress = NuCore.straddrstr "Click" CreditsGroupAddress "Back"

// field constants
let FieldAddress = NuCore.addr "Field"
let FieldGroupName = Lun.make "Group"
let FieldGroupAddress = FieldAddress @ [FieldGroupName]
let FieldGroupFileName = "Assets/OmniBlade/Groups/Field.nugroup"
let ClickFieldBackAddress = NuCore.straddrstr "Click" FieldGroupAddress "Back"

// time constants
let TimeAddress = NuCore.addr "Time"

// now we have something worth explaining. This function adds the OmniBlade title screen to
// the world.
let addTitleScreen world =

    // this adds a dissolve screen from the specified file with the given parameter
    let world_ =
        World.addDissolveScreenFromFile
            TitleGroupFileName
            TitleGroupName
            IncomingTime
            OutgoingTime
            TitleAddress
            world

    // this subscribes to the event that is raised when the Title screen's New Game button is
    // clicked, and handle the event by transitioning to the Field screen
    let world_ =
        World.subscribe
            ClickTitleNewGameAddress
            []
            (World.handleEventAsScreenTransition TitleAddress FieldAddress)
            world_

    // subscribes to the event that is raised when the Title screen's Load Game button is
    // clicked, and handle the event by transitioning to the Field screen
    let world_ =
        World.subscribe
            ClickTitleLoadGameAddress
            []
            (World.handleEventAsScreenTransition TitleAddress LoadGameAddress)
            world_

    // subscribes to the event that is raised when the Title screen's Credits button is
```

```
    // clicked, and handle the event by transitioning to the Field screen
    let world_ =
        World.subscribe
            ClickTitleCreditsAddress
            []
            (World.handleEventAsScreenTransition TitleAddress CreditsAddress)
            world_

    // subscribes to the event that is raised when the Title screen's Exit button is clicked,
    // and handle the event by exiting the game
    World.subscribe ClickTitleExitAddress [] World.handleEventAsExit world_

// pretty much the same as above, but for the Load Game screen
let addLoadGameScreen world =

    let world' =
        World.addDissolveScreenFromFile
            LoadGameGroupFileName
            LoadGameGroupName
            IncomingTime
            OutgoingTime
            LoadGameAddress
            world

    World.subscribe
        ClickLoadGameBackAddress
        []
        (World.handleEventAsScreenTransition LoadGameAddress TitleAddress)
        world'

// and so on...
let addCreditsScreen world =

    let world' =
        World.addDissolveScreenFromFile
            CreditsGroupFileName
            CreditsGroupName
            IncomingTime
            OutgoingTime
            CreditsAddress
            world

    World.subscribe
        ClickCreditsBackAddress
        []
        (World.handleEventAsScreenTransition CreditsAddress TitleAddress) world'

// and so on.
let addFieldScreen world =

    let world' =
        World.addDissolveScreenFromFile
            FieldGroupFileName
            FieldGroupName
            IncomingTime
            OutgoingTime
            FieldAddress
            world

    World.subscribe
        ClickFieldBackAddress
        []
        (World.handleEventAsScreenTransition FieldAddress TitleAddress)
        world'

// here we create the OmniBlade world in a callback from the World.run function.
let tryCreateOmniBladeWorld sdlDeps extData =

    // our custom game dispatcher here is OmniGameDispatcher
```

```
let gameDispatcher = OmniGameDispatcher () :> obj

// we use the World.tryCreateEmptyWorld as a convenience function to create an empty world
// that we will transform to create the OmniBlade world.
let optWorld = World.tryCreateEmptyWorld sdlDeps gameDispatcher extData
match optWorld with
| Left _ as left -> left
| Right world ->

    // specify how to create the sprite used for the game's splash screen
    let splashScreenSprite =
        { SpriteAssetName = Lun.make "Image5"
          PackageName = Lun.make "Default"
          PackageFileName = "AssetGraph.xml" }

    // add to the world a splash screen that automatically transitions to the Title screen
    // when finished
    let world_ =
        World.addSplashScreenFromData
            (World.transitionScreenHandler TitleAddress)
            SplashAddress
            IncomingTimeSplash
            IdlingTime
            OutgoingTimeSplash
            splashScreenSprite
            world

    // specify the song to play for the duration of the game
    let gameSong =
        { SongAssetName = Lun.make "Song"
          PackageName = Lun.make "Default"
          PackageFileName = "AssetGraph.xml" }

    // create a message to the audio system to play said song
    let playGameSong =
        PlaySong
            { Song = gameSong
              FadeOutCurrentSong = true }

    // add the message to the audio message queue
    let world_ = { world_ with AudioMessages = playGameSong :: world.AudioMessages }

    // add our UI screens to the world
    let world_ = addTitleScreen world_
    let world_ = addLoadGameScreen world_
    let world_ = addCreditsScreen world_
    let world_ = addFieldScreen world_

    // transition the world to splash screen
    let world_ = World.transitionScreen SplashAddress world_

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

As you might notice, 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, edge-case-ridden, 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]
                                |
                                |--> ScreenTransition(In)
                                |
                                |--> ScreenTransition(Out)
```

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.

Further, anyone 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 type means 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 holds all the other information needed to execute a game such as the rendering context, audio context, their related message queues (most on this later), a purely-functional event system (far superior to .Net's built-in mutable event system), and all other types of dependencies and configuration values. When you want something in your game to change, you start here and work your way down.

Screen

Screens are precisely what they sound like – a way to implement a single 'screen' of interaction in your game. In Nu's mental model, a game is nothing more than a series of interactive screens to be traversed (like a graph). How screens transition from one to another is specified in code. In fact, we've already seen the code that does that in the OmniBlade.OmniFlow.addTitleScreen function a few pages above.

ScreenTransition(s)

Screen transitions in other engine, like their UI, are typically tacked on, if present at all. However, Nu knows that no game wants to move from one screen to another without some sort of pleasant graphical transition sequence.

There are two ScreenTransitions per screen; one describes how the screen transitions in from other screens, and the other describes how it transition out to other screens.

Group

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

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 differs a button entity from a block entity, though? Each entity picks up its unique attributes from its XType. What is an XType? Well, it's a little complicated, so we'll touch on that later!