Nu Game Engine

The world's first practical, pure functional game engine!

Bryan Edds, 2014

Table of Contents

Table of Contents ..... - 2 -
What's It All About? ..... - 3 -
  Simple ..... - 3 -
  Purely-Functional(ish) ..... - 3 -
  2d Game Engine ..... - 3 -
  F# ..... - 3 -
Getting Started ..... - 4 -
  Creating your own Nu game Project ..... - 4 -
  Basic Nu Start-up Code ..... - 7 -
What is NuEdit? ..... - 9 -
BlazeVector ..... - 14 -
The Game Engine ..... - 18 -
  World ..... - 18 -
  Screen ..... - 18 -
  Group ..... - 18 -
  Entity ..... - 19 -
Game Engine Details ..... - 19 -
  Addresses ..... - 19 -
  Transformations ..... - 19 -
  The Purely-Functional Event System ..... - 20 -
  Xtensions ..... - 20 -
    Understanding the Xtension Type ..... - 21 -
    How Nu uses Xtensions in practice ..... - 21 -
  Dispatchers ..... - 22 -
  Facets ..... - 23 -
More on BlazeVector ..... - 26 -
  Bullets and the BulletDispatcher ..... - 26 -
    Bullets in NuEdit ..... - 27 -
    The code behind the bullets ..... - 29 -
    The Register override ..... - 31 -
    The tickHandler ..... - 31 -
    The collisionHandler ..... - 32 -
  Enemies and the EnemyDispatcher ..... - 32 -
    Enemies in NuEdit ..... - 32 -
More Engine Details ..... - 35 -
  Assets and the AssetGraph ..... - 35 -
  Serialization and Overlays ..... - 36 -
  Subsystems and Message Queues ..... - 38 -

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 few 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 some 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 Nu, you are empowered to write in the pure-functional style.

2d Game Engine

Nu is not a code library. It is a **game software framework**. Thus it 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" though; 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 from 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 user 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 if you need to make changes to or debug into the game engine.

Upon inspecting your clone of the repository, the first thing you might notice about it is that the repository 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 (the free **Express** version is fine). 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.

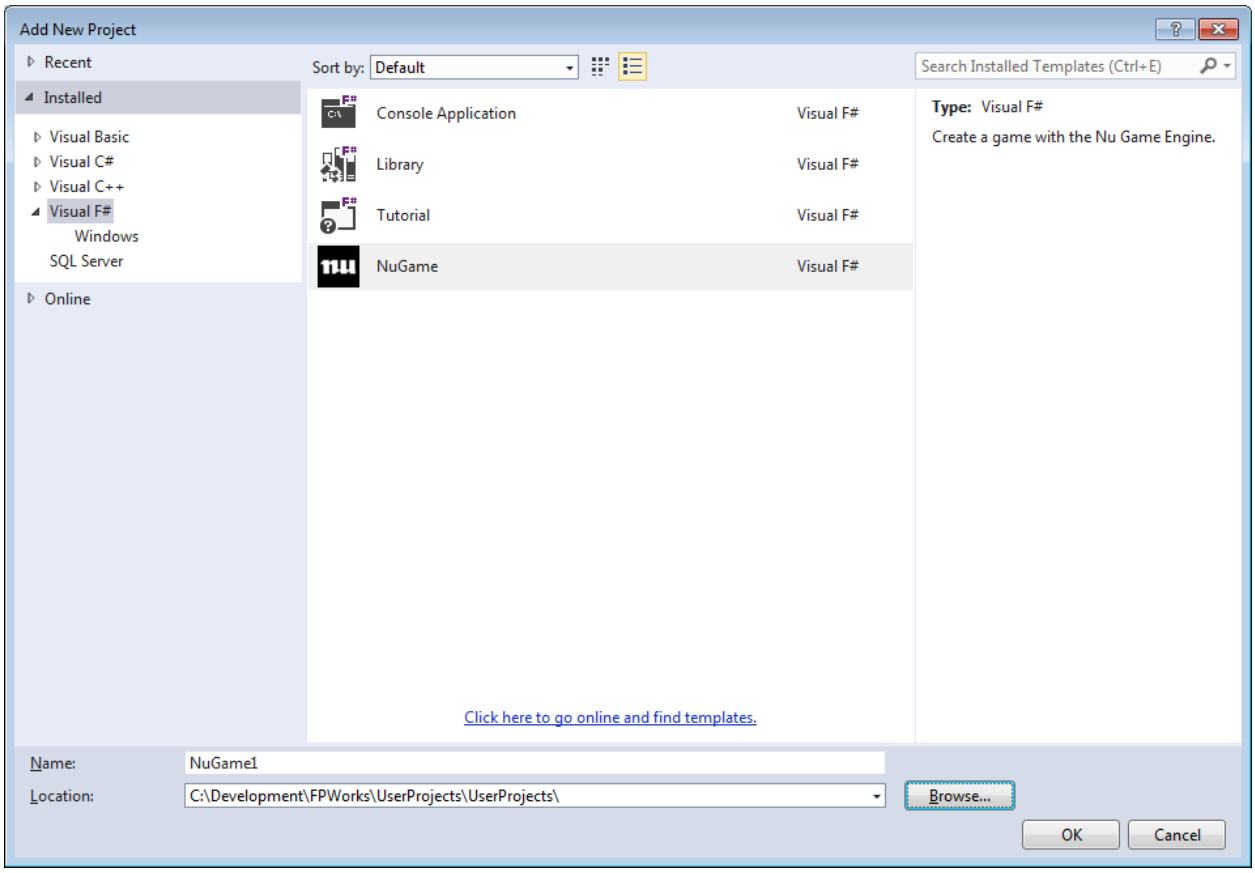
Once the solution builds successfully, ensure that the **BlazeVector** project is set as the **StartUp** project, and then run the game by pressing the **|> Start** button in Visual Studio.

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.

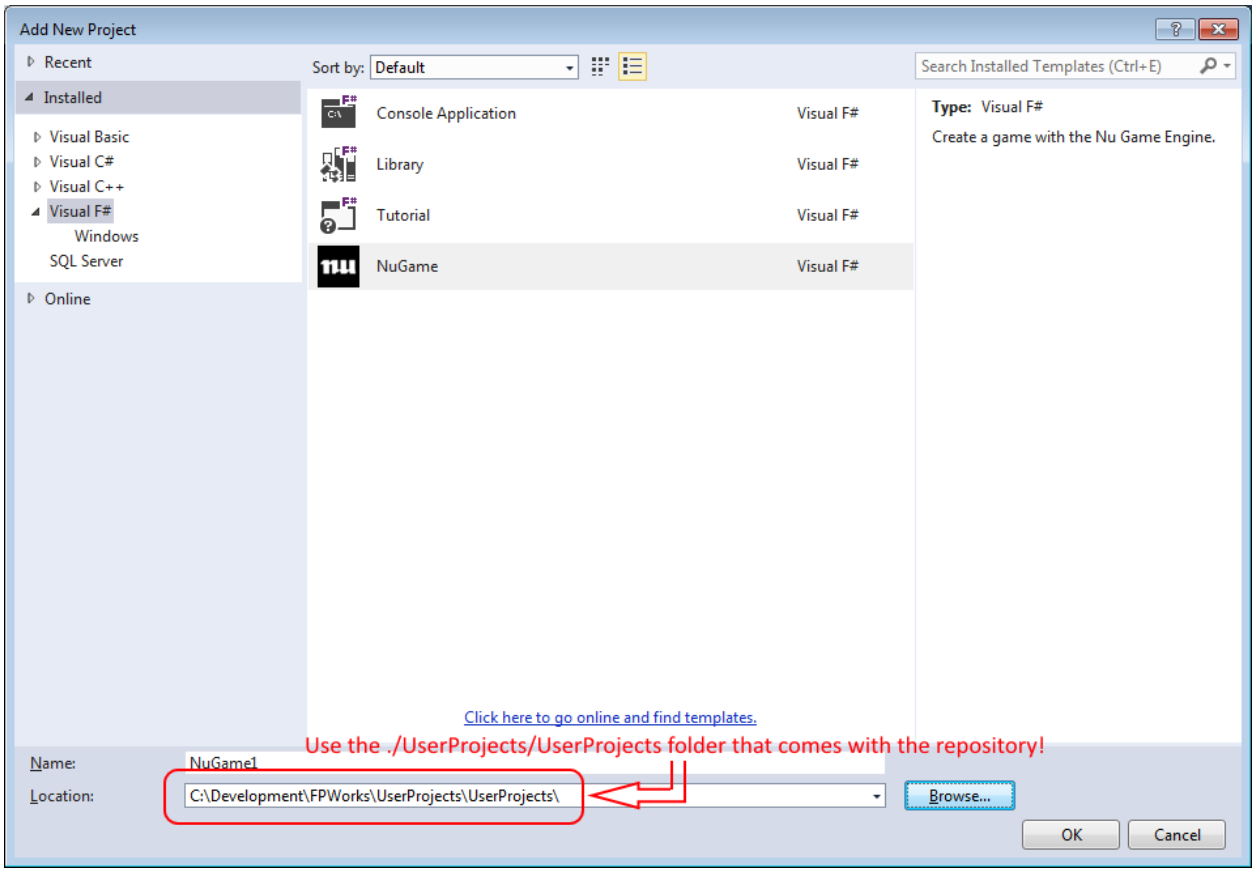
Back in the BlazeVector solution in Visual Studio, click **File -> Add -> New Project**. Under the **Visual F# category**, select the **NuGame** template like so –



- and enter the name of your game in the **Name** text box.

WARNING: Do NOT create a project by clicking **File -> New Project…**! This will create a new project in its own solution, separate from the current one, and that is NOT what you want ☺

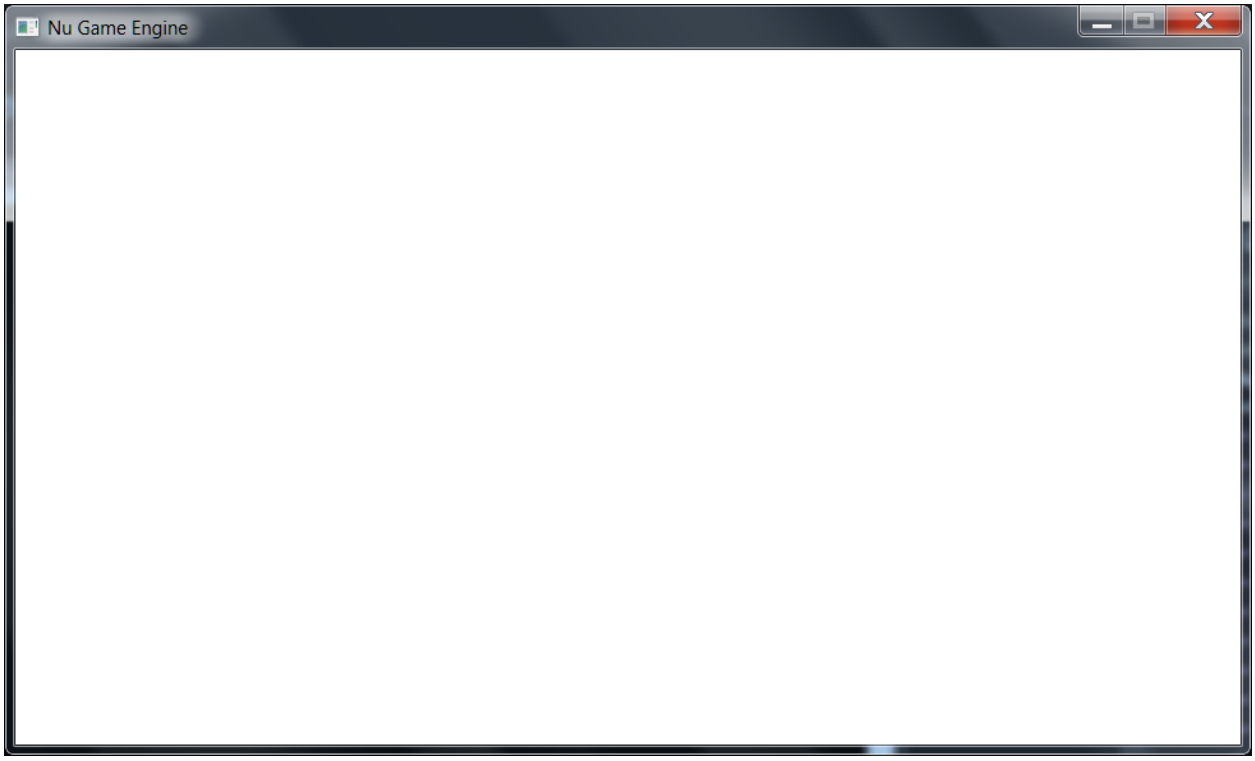
IMPORTANT: Next, set the Location field to the **./UserProjects/UserProjects** folder like so –



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 the **OK** button to create the project. Now you can try building and running the new project by setting it as the **StartUp** project and then pressing the **|> Start** button.

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



Though this is not yet an interesting program, a look at the initial code behind it should give us an idea of how to proceed.

Basic Nu Start-up Code

Here's the main code presented with comments -

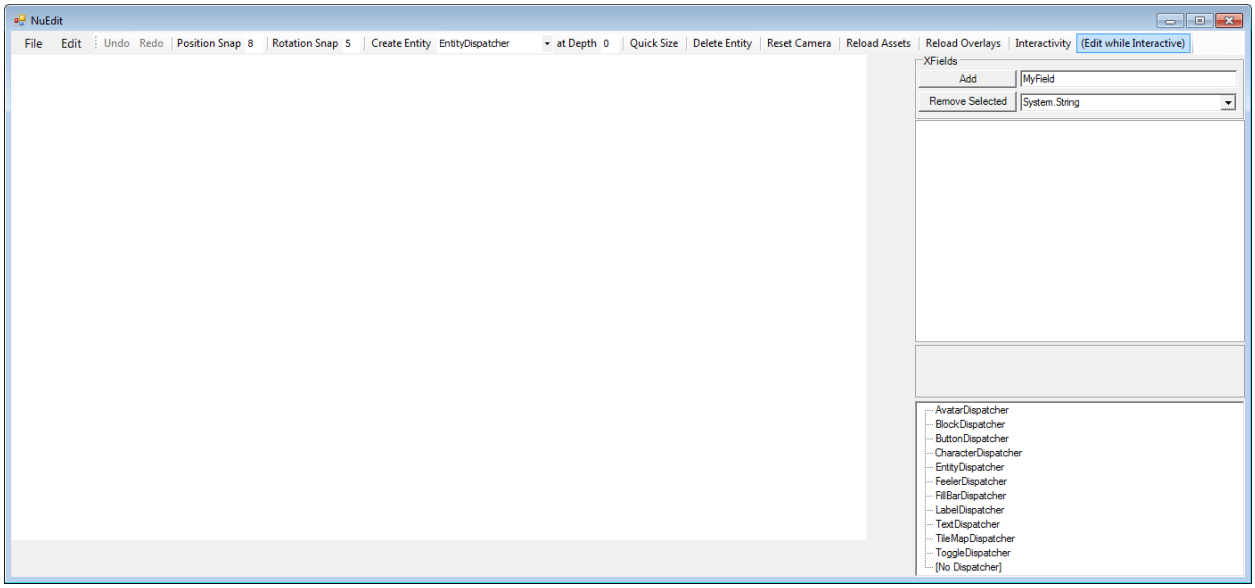
```
namespace NuGame1
open SDL2
open Prime
open Nu
open Nu.Constants
module Program =

    // this is a factory that creates user-defined components such as dispatchers of various sorts
    // and facets. Currently, there are no overrides for its factory methods since there are no
    // user-defined dispatchers defined yet for this project.
    type NuGame1ComponentFactory () =
        inherit UserComponentFactory ()

    // this the entry point for the your Nu 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 =

        // A component factory is the means by which user-defined dispatchers and facets are
        // made available for creation by the engine, as well as by NuEdit. Note that any
        // user-defined game dispatcher returned from a user-defined component factory will
        // be used as your game's dispatcher.
        let userComponentFactory = NuGame1ComponentFactory ()

        // 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
        // component factory above, a boolean that protects from a Farseer physics bug but
        // slows down the engine badly, 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.tryMake sdlDeps userComponentFactory GuiAndPhysicsAndGamePlay false ()

    // 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 your game. 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 editing tool. 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!*

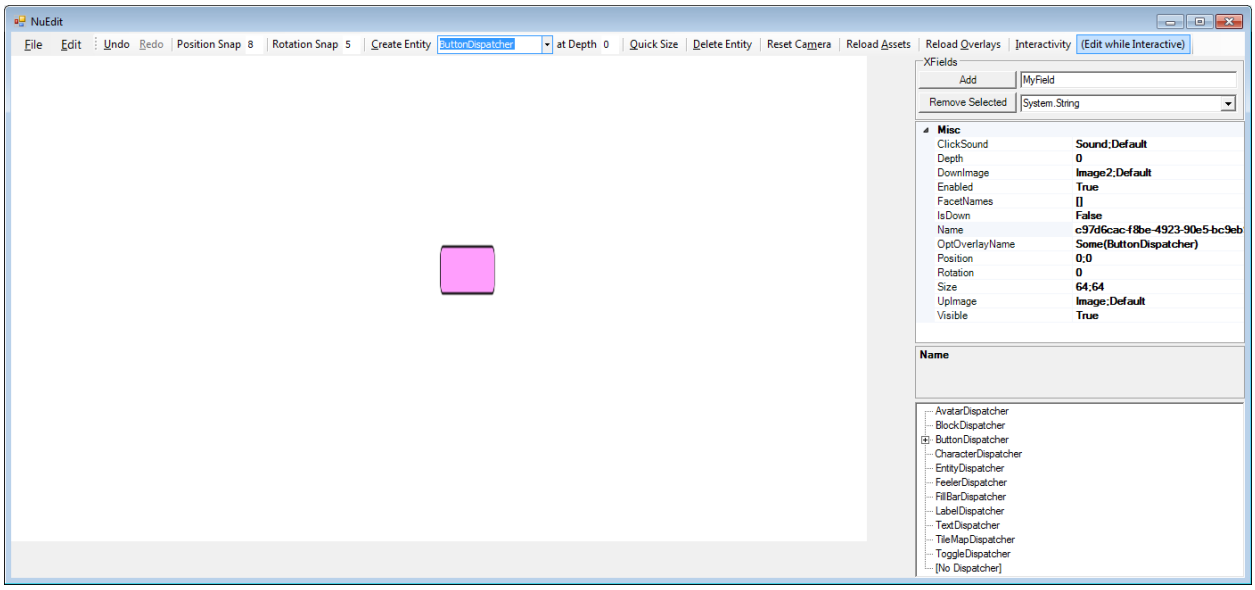
Run NuEdit by setting the **NuEdit project** as the StartUp Project in Visual Studio, and then running.

You'll instantly notice an **Open File dialog** appear from which you are instructed to "Select your game's executable file…" If you select a.NET assembly (or executable) that contains a subclass of the UserComponentFactory, the dispatchers and facets it creates will be available for use in the editor. If you cancel the dialog, you get only what comes with Nu.

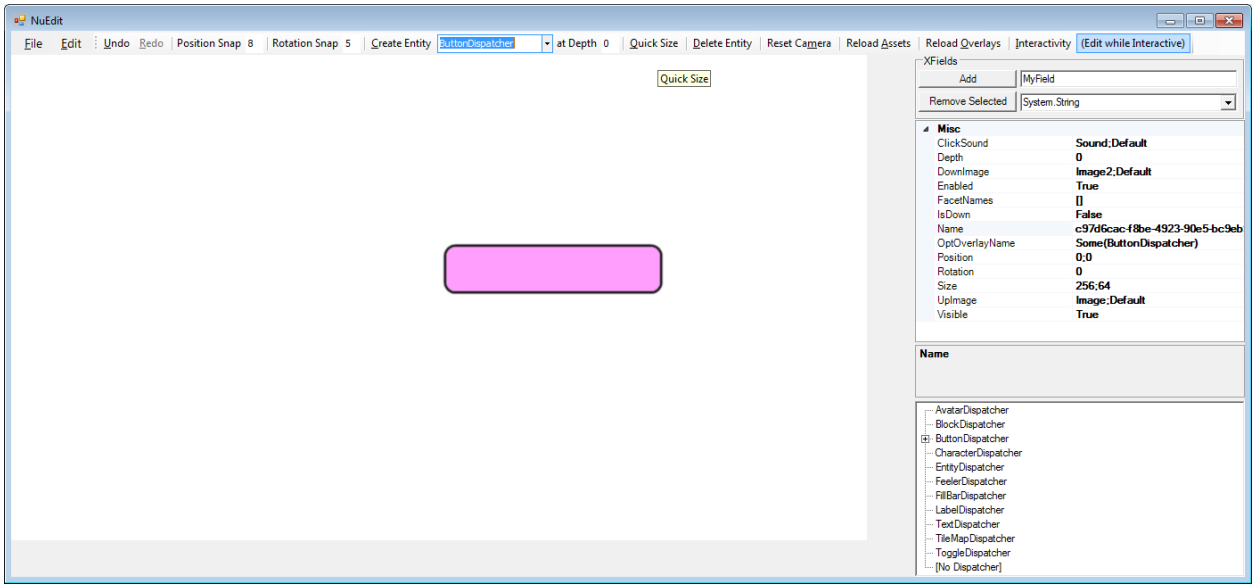
"Just what is a dispatcher / facet, anyway?" you might ask. Good question! However, we will wait to explain them detail later. For now, just know that they are used to define and compose custom simulants for your game!

Here we will just cancel and play with the dispatchers / facets that come out-of-the-box.

First, let's create a blank button in NuEdit by selecting **ButtonDispatcher** from the combo box to the right of the **Create Entity** button, and then pressing the **Create Entity** button.



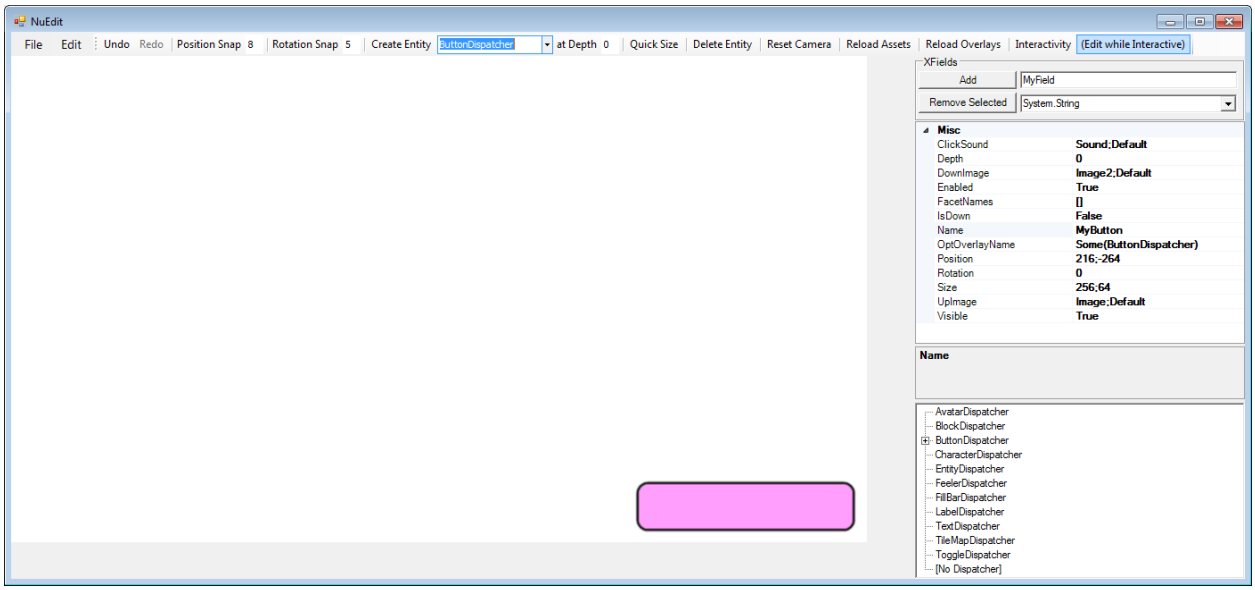
You'll notice a squished button appear in the middle of the editing panel. By default, most entities are created with a size of 64x64. Fortunately, Nu gives you an easy way to resize the entity to fit the button's image by pressing the **Quick Size** button. Press it now.



We have a full-sized button! Now notice that the property grid on the right has been populated with the button's fields. These fields can be edited to change the button however you like. For a button that will be used to control the game's stat, the first thing you will want to do is to give it an appropriate name. Do so by double-clicking the **Name** field, deleting the contents, and then entering 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. You can also right-click and entity for additional operations via a context menu.

Here is the renamed the button after having moved it to the bottom right of the screen –



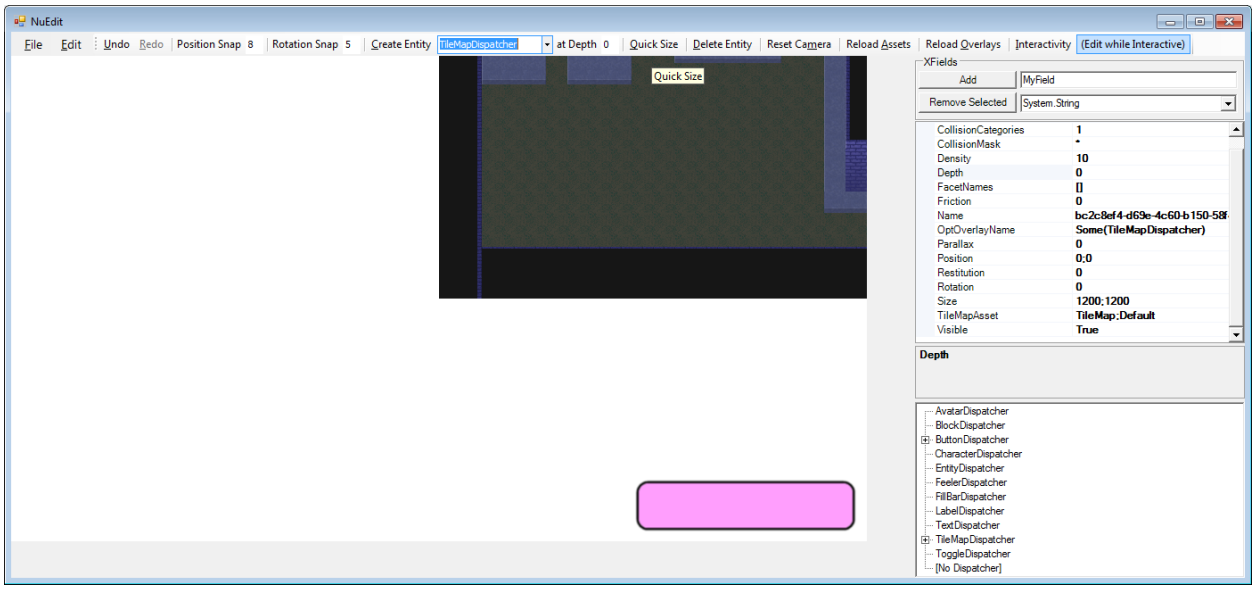
Notice you have the full power of **Undo** and **Redo**. Nonetheless, you should still save your documents often in case this early version of NuEdit goes bananas on you.

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

Once you're satisfied, toggle off the **Interactivity** button to return to the non-interactive mode.

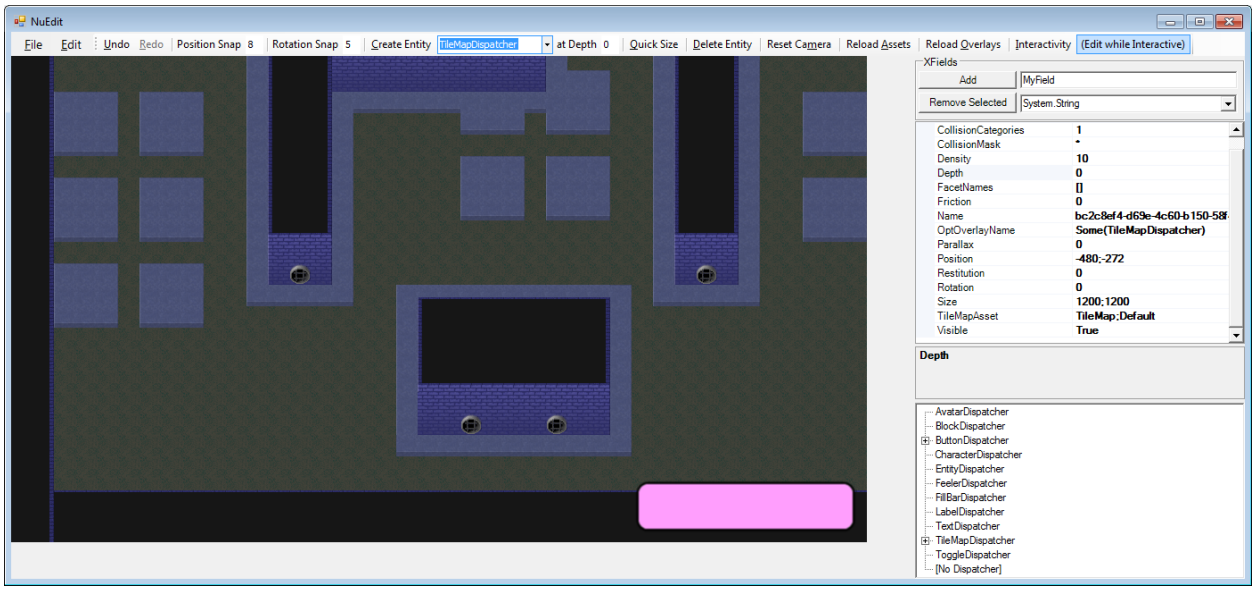
Now let's make a default tile map to play around with. BUT FIRST, we need to change the depth of our button entity so that it doesn't get covered by the new tile map. Change the value in the button's **Depth** field to **10**.

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.

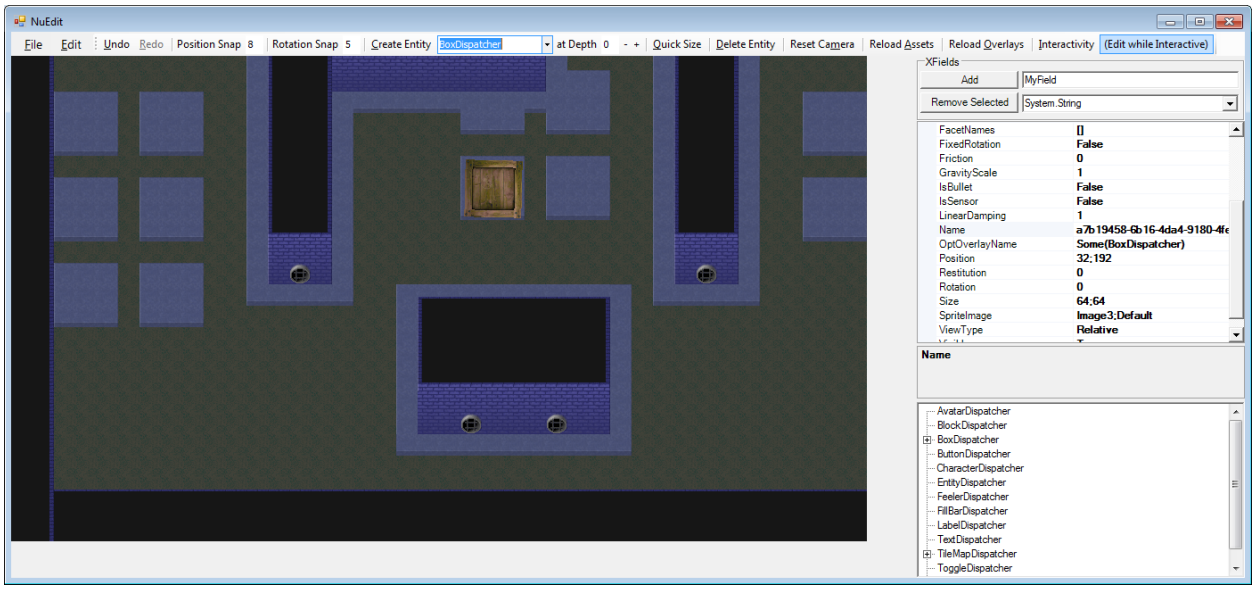
Tile maps, by the way, are created with the free tile map editor **Tiled** found at http://www.mapeditor.org/. All credit to the great chap who made and maintains it!



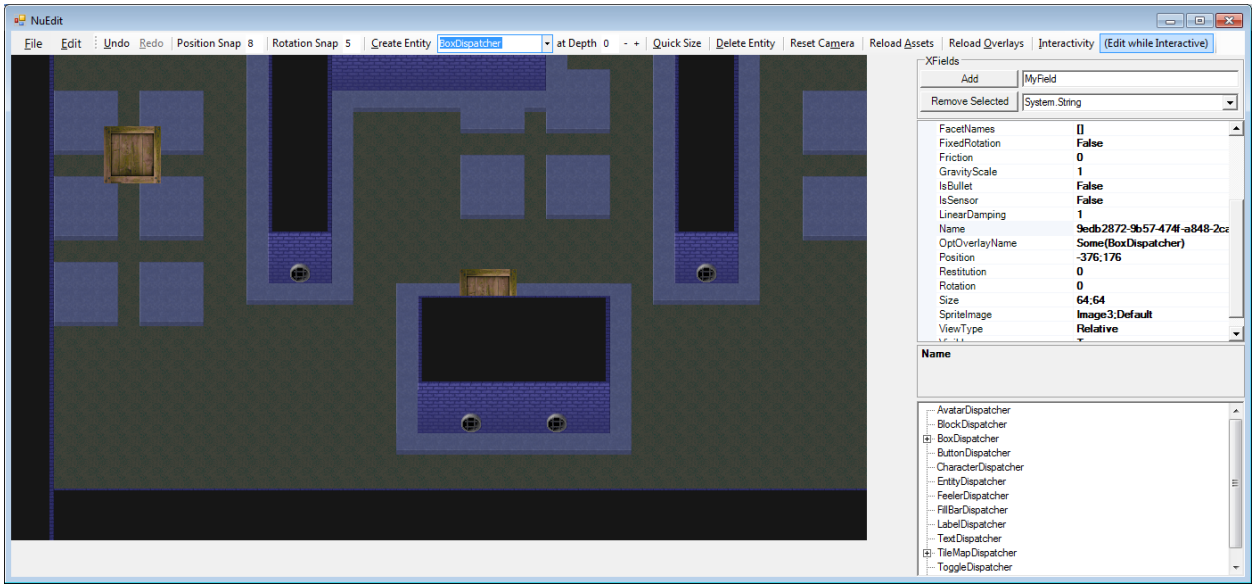
Now click and drag with the MIDDLE mouse button to change the position of the camera that is used to view the game. Check out your lovely new tile map! If your camera gets lost in space, click the **Reset Camera** button that is to the left of the **Interact** button.

Now let's create some boxes that use physics to fall down and collide with the tile map. 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 **1**. In the combo box to the right of the **Create Entity**

button, select (or type) **BoxDispatcher**, and then click the **Create Entity** button. You'll see a box that was created in the middle of the screen that falls directly down.



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



Boxes 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 the game. By studying BlazeVector's top level code, we can see how a typical Nu game is composed.

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 = !* "Splash"

    // title constants
    let TitleAddress = !* "Title"
    let TitleGroupFileName = "Assets/BlazeVector/Groups/Title.nugroup"
    let TitleGroupAddress = !* "Title/Group"
    let SelectTitleEventName = !* "Select/Title"
    let ClickTitlePlayEventName = !* "Click/Title/Group/Play"
    let ClickTitleCreditsEventName = !* "Click/Title/Group/Credits"
```

```
    let ClickTitleExitEventName = !* "Click/Title/Group/Exit"

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

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

Nothing terribly interesting except maybe the **!*** operator. It is merely an operator that converts a string to an Address, which is how Nu looks up simulants in the engine. Let's go ahead and jump to Program.fs -

```
namespace BlazeVector
open System
open SDL2
open Nu
open Nu.Constants
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 ())
            (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.Constants
open BlazeVector
open BlazeVector.BlazeConstants
module BlazeFlow =

    /// Creates BlazeVector-specific components (dispatchers and facets).
    /// Allows BlazeVector simulation types to be created in the game as well as in NuEdit.
    type BlazeComponentFactory () =
        inherit UserComponentFactory ()

        // make our game-specific entity dispatchers...
        override dispatcher.MakeEntityDispatchers () =
            Map.ofList
                [typeof<BulletDispatcher>.Name, BulletDispatcher () :> EntityDispatcher
                 typeof<PlayerDispatcher>.Name, PlayerDispatcher () :> EntityDispatcher
                 typeof<EnemyDispatcher>.Name, EnemyDispatcher () :> EntityDispatcher]

        // make our game-specific group dispatchers...
        override dispatcher.MakeGroupDispatchers () =
            Map.ofList
                [typeof<StagePlayDispatcher>.Name, StagePlayDispatcher () :> GroupDispatcher]

        // make our game-specific screen dispatchers...
        override dispatcher.MakeScreenDispatchers () =
            Map.ofList
                [typeof<StageScreenDispatcher>.Name, StageScreenDispatcher () :> ScreenDispatcher]

        // make our game-specific game dispatchers...
        override dispatcher.MakeGameDispatchers () =
            Map.ofList
                [typeof<BlazeVectorDispatcher>.Name, BlazeVectorDispatcher () :> GameDispatcher]

        // currently we have no game-specific facets to create, so no need to override MakeFacets.

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

    // this function handles playing the stage
    let handlePlayStage _ world =
        let oldWorld = world
        let world = World.fadeOutSong DefaultTimeToFadeOutSongMs world
        let stageScreen = World.getScreen StageAddress world
        match World.tryTransitionScreen StageAddress stageScreen world with
        | Some world -> (Propagate, world)
        | None -> (Propagate, oldWorld)

    // 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 = snd <| 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 = snd <| 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 = snd <| 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 userState =

        // create our game's component factory
        let blazeComponentFactory = BlazeComponentFactory ()

        // we use World.tryMake to create an empty world that we will transform to create the
        // BlazeVector world
        let optWorld = World.tryMake sdlDeps blazeComponentFactory GuiAndPhysicsAndGamePlay false userState
        match optWorld with
        | 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 (splashScreen, world) = World.addSplashScreenFromData TitleAddress SplashAddress typeof<ScreenDispatcher>.Name
IncomingTimeSplash IdlingTime OutgoingTimeSplash splashScreenImage world

            // play a neat sound effect, select the splash screen, and we're off!
            let world = World.playSound NuSplashSound 1.0f world
            let world = snd <| World.selectScreen SplashAddress splashScreen world
            Right world

        // propagate error
        | Left _ as left -> left
```

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 we need to learn more about the game engine itself before delving into them.

As a final note, you might notice that 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? 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 needed by imperative undo / redo systems.

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'). Each **Screen** may contain multiple **Groups** that may in turn each contain multiple **Entities**.

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 say a box with physics and a UI button.

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 with the **World** value and work your way inward (using handy functions set as World.setEntity along the way).

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

Groups represent logical 'groupings' of entities. NuEdit builds one group of entities at a time. At run-time in your game, multiple groups can be 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 boxes. What differentiates a button entity from a box entity, though? Each entity picks up its unique attributes from its **dispatcher**. What is a dispatcher? Well, it's a little complicated, so we'll touch on that slightly later! Please be patient ☺

Game Engine Details

Addresses

You may be wondering how to access at run-time a specific entity that was created in the editor and loaded from an XML file. Accessing entities done by passing its **address** to the **World.getEntity** function. Each entity has an address of the form **ScreenName/GroupName/EntityName**, where **ScreenName** is the name that is given to the containing screen, **GroupName** is the name given to the containing group, and **EntityName** is the name given to the entity (such as in the editor). Remember how we changed the **Name** field of the button object that we created to **MyButton** earlier in **NuEdit**? That's what we're talking about, and the entity's name is just the last part of its address.

Similar queries exist for both groups and screens.

Transformations

Given all this, how do we actually make transformations to a given entity in the world? Since Nu is a purely-functional game engine, we can't just change the field of a given entity in-place...

First, we need to find the entity in the world that we want to transform. Second, we have to transform it, and thirdly we place the transformed value unto a new copy of the containing world.

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

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

This will return an entity placed at the given address. Now let's transform that button, say, by toggling its **Enabled** field -

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

Finally, let's place the transformed value unto a new copy of the world using the World.setEntity function –

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

So, maybe I hear you saying, "I thought functional programming was supposed to be concise! This is a lot of work just to transform a field!" Maybe. The truth is, functional programming is generally quite succinct, especially when build language interpreters. But when it comes to transforming simulation graphs, it's not quite as handy. However, we make the trade-off for functional programming to achieve other more important goals such as correctness, and these properties are still preserved even in the face of simulations – or perhaps *especially* in the face of simulations!

The Purely-Functional Event System

Because the event system that F# provides out of the box is inherently mutating / impure, I invented a simple, purely-functional event system for Nu.

Subscriptions are created by invoking the World.subscribe function, and destroyed using the World.unsusbscribe function. Since subscriptions are to address rather than particular simulants, you can subscribe to any address regardless of whether there exists a simulant there or not!

Additionally, there is a function that subscribes to events only for the lifetime of the subscriber. It is World.observe. You will likely be using this more often than the other two functions as it more compactly provides the desired behavior.

Events, like simulants, also have addresses. For example, say there is a button at address **MyGame/MyScreen/MyGroup/MyButton**. To subscribe to its **Click** event, you provides the following as the subscription address – **Click/MyGame/MyScreen/MyGroup/MyButton**. The nature of the event is always prepended to the address of the event source. To subscribe to input events like the left mouse button being pressed, you don't need a specific event source at all – just susbscribe to **Down/Mouse/Left**! This is the same for other unsourced, global events like **Down/KeyboardKey** input and **Tick**.

Since there is currently no guide to what all events take place and their address specifications, you have to consult **NuConstants.fs** in the **Nu** project to figure out how to subscribe to the various events. Fortunately, it's mostly common sense.

Using this same addressing scheme, you can make and publish your own custom events as well!

TODO: more detail!

TODO: comprehensive event guide!

Xtensions

Xtensions are a key enabling technology in Nu. Xtensions allow the **Game**, **Screen**, **Group**, and **Entity** types (hereafter collectively known as the **simulation types**) to be extended by the end-user in a purely-functional way. This extensibility mechanism is the key creating your own simulation types.

Xtensions, very simply, allow data fields to be added to a simulation type at run-time. We can see how they are used to provide these capabilities to the Entity type by studying the type here –

```
/// The type around which the whole game engine is based! Used in combination with dispatchers
/// to implement things like buttons, avatars, blocks, and things of that sort.
and [<CLIMutable; StructuralEquality; NoComparison>] Entity =
    { Id : Guid
      Name : string
      Position : Vector2
      Depth : single
      Size : Vector2
      Rotation : single
      Visible : bool
      ViewType : ViewType
      DispatcherNp : EntityDispatcher
      FacetNames : string list
      FacetsNp : Facet list
      OptOverlayName : string option
      Xtension : Xtension

    static member (?) (this : Entity, memberName) =
```

```
        Xtension.(?) (this.Xtension, memberName)

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

The three aspects that give entities "xtensibility" is the **Xtension** field and the **(?)** and **(?<-)** operator overloads. We will discuss more about the overloads in just a moment.

Understanding the Xtension Type

Perhaps the most efficient way to exemplify the usage of an Xtension is by discussing its unit tests. In the tests, Be aware that in the following tests Xtensions are exercised in isolation, though of course the engine uses them by embedding them in a type as above. 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. 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 **'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 the Xtension's members are not usually accessed directly, but only accessed 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 the forwarding operators 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?Enabled <- value
```

- which when used in practice look like this –

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

This is to allow user code to use the most stringent level of typing possible even though such members are in actuality dynamic!

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

There is a trade-off however that annoys me to no end. In order to capture full static for new fields, the end-user must create extension functions like the one for the Enabled field above. Perhaps if we had macros in F#, such code could be automatically generated for you, but alas, we have it not. Of course, you could just use the dynamic operators to access your user-defined fields dynamically, but this is not idiomatic.

Dispatchers

A **dispatcher** is a stateless object that allows you to specify the behavior of a simulation type. Dispatchers are a simple implementation of a technique that harkens back to the **Strategy Pattern** of OOP yore, but are totally stateless.

Since simulation types are F# records for functional purity, and because we can't extend records, we need a way to define custom behavior for simulation types such as entities. You might have noticed a field in the Entity type defined as –

```
DispatcherNp : EntityDispatcher
```

This is the field that is configured with the appropriate dispatcher object by the engine. You can see its type is of **EntityDispatcher**, which is defined as thus –

```
/// The default dispatcher for entities.
and EntityDispatcher () =

    static member FieldDefinitions =
        [define? Position Vector2.Zero
         define? Depth 0.0f
         define? Size DefaultEntitySize
         define? Rotation 0.0f
         define? Visible true]

    abstract member Register : Address * Entity * World -> Entity * World
    default dispatcher.Register (_, entity, world) = (entity, world)

    abstract member Unregister : Address * Entity * World -> Entity * World
    default dispatcher.Unregister (_, entity, world) = (entity, world)

    abstract member PropagatePhysics : Address * Entity * World -> World
    default dispatcher.PropagatePhysics (_, _, world) = world

    abstract member GetRenderDescriptors : Entity * World -> RenderDescriptor list
```

```
        default dispatcher.GetRenderDescriptors (_, _) = []

        abstract member GetQuickSize : Entity * World -> Vector2
        default dispatcher.GetQuickSize (_, _) = Vector2.One

        abstract member GetPickingPriority : Entity * World -> single
        default dispatcher.GetPickingPriority (entity, _) = entity.Depth
```

The **FieldDefinitions** are a special static member that configures the properties of the entity that the dispatcher has been attached to. We'll talk more about this later.

The **Register** method allows you to customize what happens to the entity (and the world) when it is added to the world. **Unregister** allows you to customize what happens when it is removed. **PropagatePhysics** describes how you would like to propagate changes in an entity's physics properties. For the semantics of this, it is best to use existing code as an example. **GetRenderDescriptors** is what must be implemented if you have some custom rendering that you want to implement. **GetQuickSize** introspects into an entity to get its optimal size inside of NuEdit. **GetPickingPriority** tells the entity picker in the editor the order in which it is in line for picking.

All these overrides are available for you to customize your entity's behavior. But that's not the only way…

Facets

A **facet** implements a single, composable behavior that can be assigned to an entity. Like a dispatcher, a facet is a complete stateless object with override-able methods. Many of its method match the shape of an EntityDispatcher's as well. Let's take a look at the definition and use of one of Nu's most basic facets now –

```
[<AutoOpen>]
module SpriteFacetModule =

    type Entity with

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

    type SpriteFacet () =
        inherit Facet ()

        static member FieldDefinitions =
            [define? SpriteImage { ImageAssetName = "Image3"; PackageName = "Default"}]

        override facet.GetRenderDescriptors (entity : Entity, world) =
            if entity.Visible && Camera.inView3 entity.ViewType entity.Position entity.Size world.Camera then
                [LayerableDescriptor
                    { Depth = entity.Depth
                      LayeredDescriptor =
                        SpriteDescriptor
                            { Position = entity.Position
                              Size = entity.Size
                              Rotation = entity.Rotation
                              ViewType = entity.ViewType
                              OptInset = None
                              Image = entity.SpriteImage
                              Color = Vector4.One }}]
            else []

        override facet.GetQuickSize (entity : Entity, world) =
            let image = entity.SpriteImage
            match Metadata.tryGetTextureSizeAsVector2 image.ImageAssetName image.PackageName
world.State.AssetMetadataMap with
            | Some size -> size
            | None -> DefaultEntitySize
```

As you may see, the **SpriteFacet** is used to define simple sprite rendering behavior for an entity.

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

```
[<AutoOpen>]
module RigidBodyFacetModule =

    type Entity with

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

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

    type RigidBodyFacet () =
        inherit Facet ()

        static let getBodyShape (entity : Entity) =
            Physics.evalCollisionExpression (entity.Size : Vector2) entity.CollisionExpression

        static member FieldDefinitions =
            [variable? MinorId <| fun () -> Core.makeId ()
             define? BodyType Dynamic
             define? Density NormalDensity
             define? Friction 0.0f
             define? Restitution 0.0f
             define? FixedRotation false
             define? LinearDamping 1.0f
             define? AngularDamping 1.0f
             define? GravityScale 1.0f
             define? CollisionCategories "1"
             define? CollisionMask "*"
             define? CollisionExpression "Box"
             define? IsBullet false
             define? IsSensor false]

        override facet.RegisterPhysics (address, entity, world) =
            let bodyProperties =
                { Shape = getBodyShape entity
                  BodyType = entity.BodyType
                  Density = entity.Density
                  Friction = entity.Friction
                  Restitution = entity.Restitution
                  FixedRotation = entity.FixedRotation
                  LinearDamping = entity.LinearDamping
                  AngularDamping = entity.AngularDamping
```

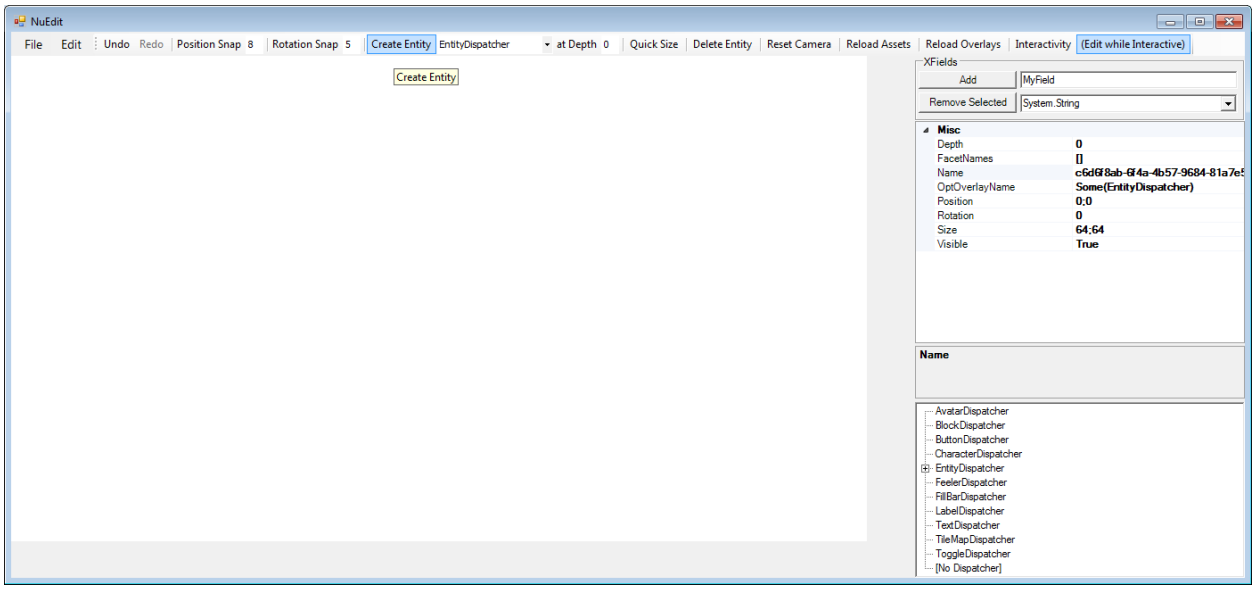
```
                GravityScale = entity.GravityScale
                CollisionCategories = Physics.toCollisionCategories entity.CollisionCategories
                CollisionMask = Physics.toCollisionCategories entity.CollisionMask
                IsBullet = entity.IsBullet
                IsSensor = 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

        override facet.UnregisterPhysics (_, entity, world) =
            World.destroyBody (Entity.getPhysicsId entity) world

        override facet.PropagatePhysics (address, entity, world) =
            let world = facet.UnregisterPhysics (address, entity, world)
            facet.RegisterPhysics (address, entity, world)
```

Complex behavior for an entity dispatcher can be defined by composing together multiple facets. Here's a dispatcher that combines the SpriteFacet and RigidBodyFacet facets at compile-time -

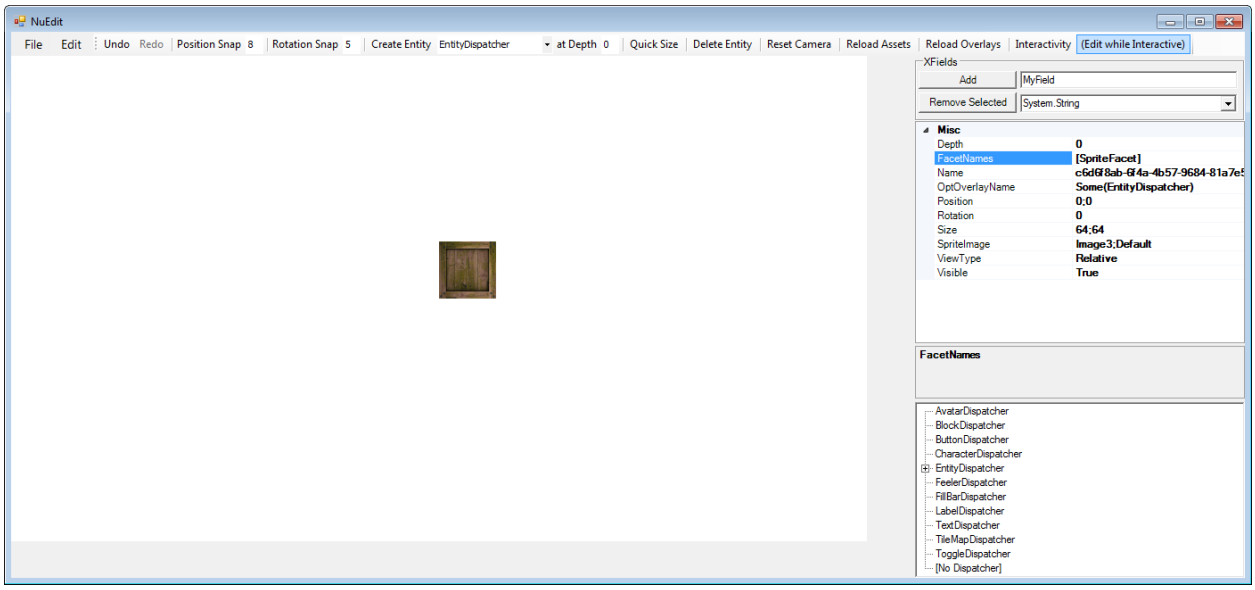
```
[<AutoOpen>]
module RigidSpriteDispatcherModule =

    type RigidSpriteDispatcher () =
        inherit EntityDispatcher ()
        static member IntrinsicFacetNames = [typeof<RigidBodyFacet>.Name; typeof<SpriteFacet>.Name]
```

Additionally, facets can be dynamically added to a removed from an entity in NuEdit simply by changing the FacetNames property. Let's take a look.

Here we just create a vanilla entity by selecting **EntityDispatcher** and pressing the **Create Entity** button –



Notice how nothing actually appears in the editing panel. This is because a plain old EntityDispatcher does not come with any rendering functionality. Let's add that now by changing its **FacetNames** field to **[SpriteFacet]** –



Not only does it now render, the additional fields needed to specify how rendering is performed are provided in the property grid (to the right). Try adding physics to the entity by changing **FacetNames** to **[SpriteFacet;RigidBodyFacet]**.

It falls away! By creating your own facets and assigning them either statically like the code above or dynamically in the editor, there's no end to the behavior you can compose!

More on BlazeVector

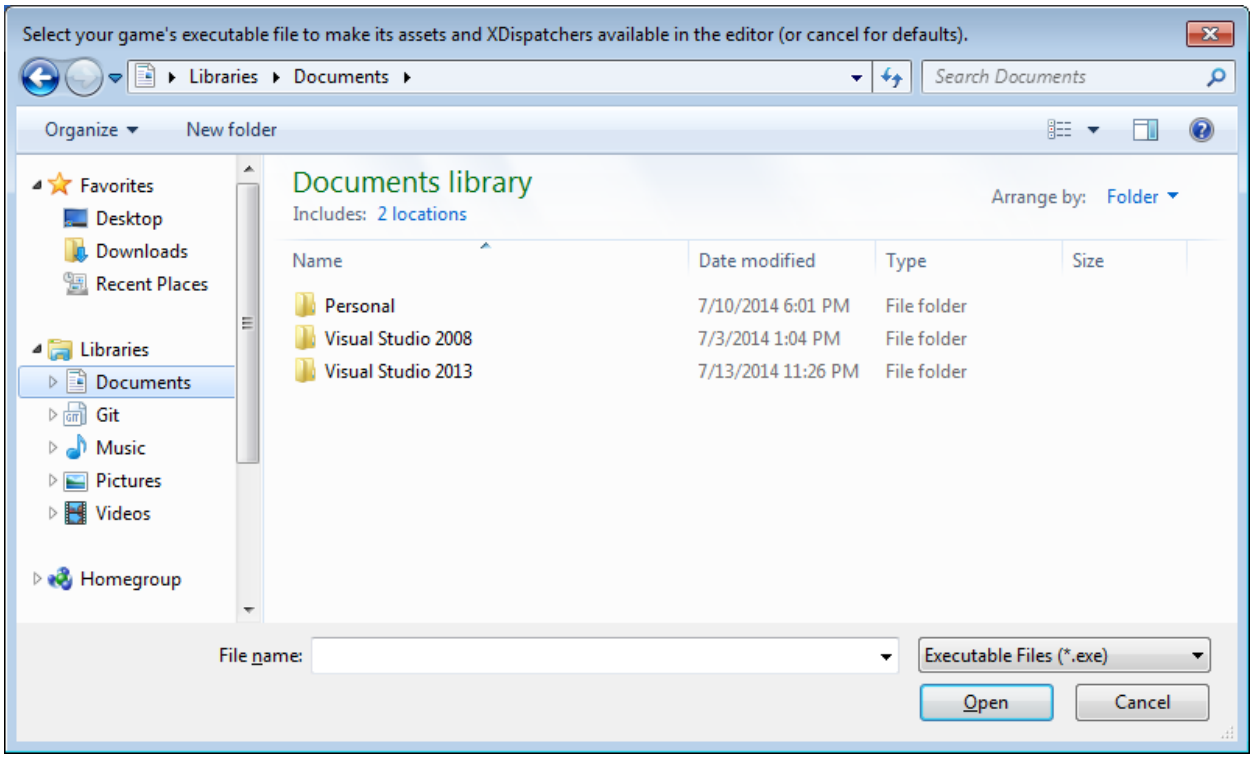
Now that we know more about the Nu Game Engine, we can explore more deeply the implementation of **BlazeVector**. In this section, we'll be loading up some of the entities used in BlazeVector in NuEdit so that we can interact with each in isolation. We'll also use that interaction as a chance to study their individual implementations.

Bullets and the BulletDispatcher

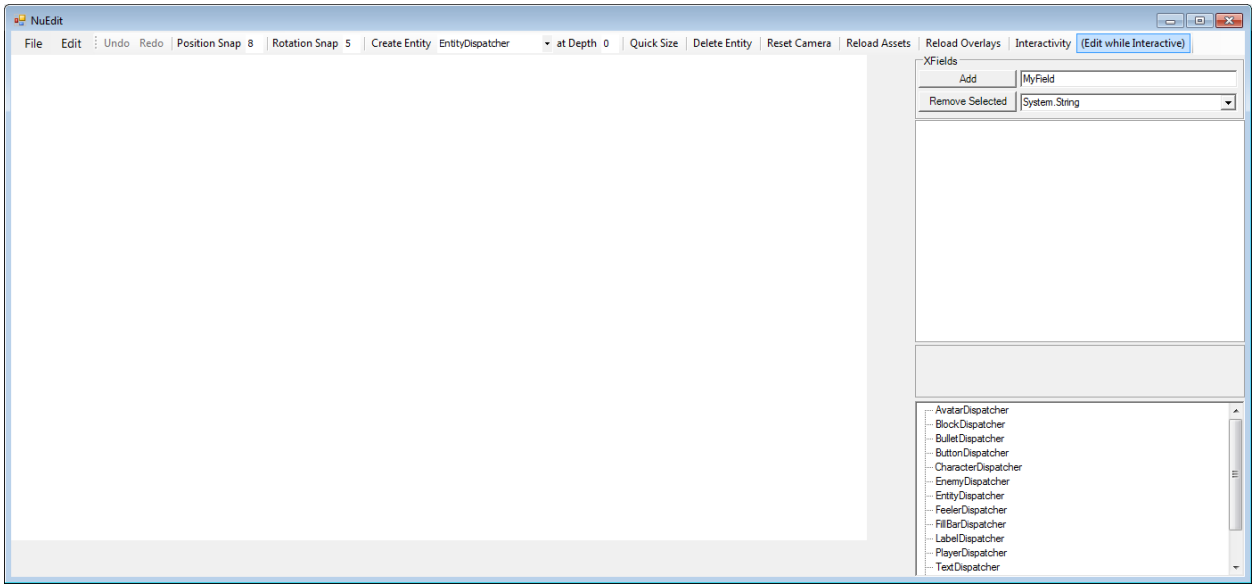
We wouldn't have much of a shooting game in BlazeVector if we didn't have bullets! Since bullets are the simplest entities defined in the BlazeVector project, let's study them first.

Bullets in NuEdit

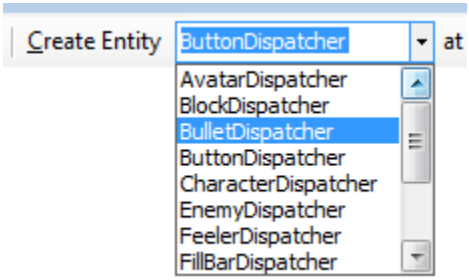
First, we'll play with a few bullet entities in the editor. If it's not already open, once again open the BlazeVector.sln, set the NuEdit project as the StartUp project, and then run it. As you know, you will see an Open File dialog appear like so –



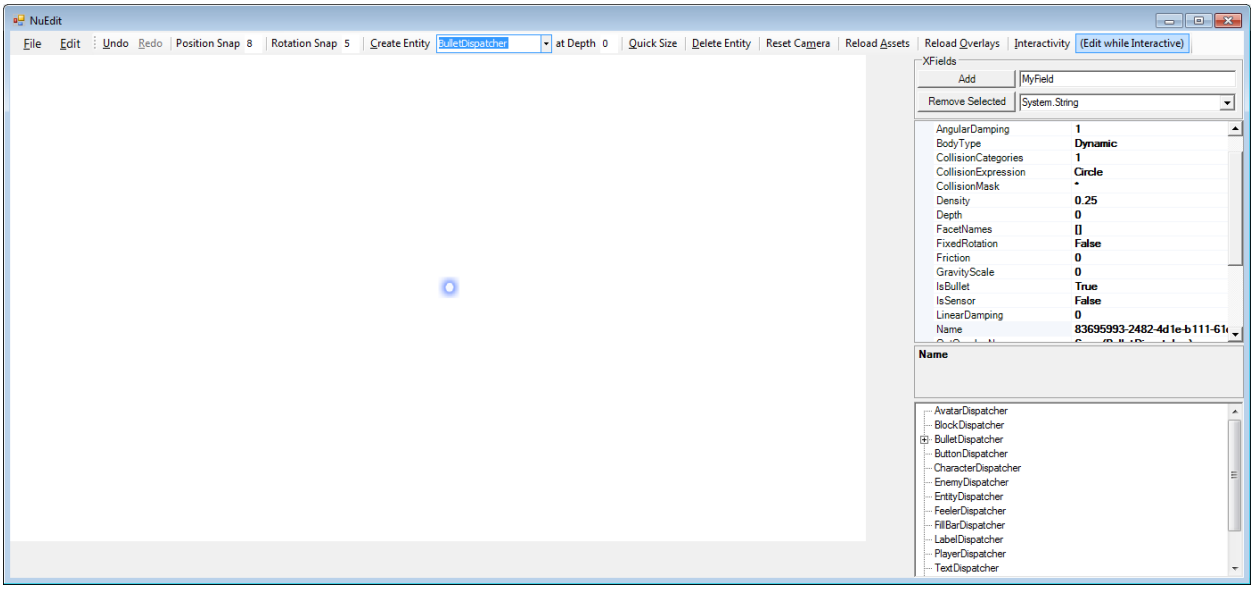
Since we need the **BulletDispatcher** from the BlazeVector.exe file, navigate the Open File dialog to the **./BlazeVector/BlazeVector/BlazeVector/bin/Debug** folder and select the **BlazeVector.exe** file. The editor will now open up as normal –



- except that if we click the drop-down button to the right of the **Create Entity** button and then scroll up, we see **BulletDispatcher** as an additional option –



Let's select the **BulletDispatcher** option and click **Create Entity** to create a bullet entity like so –

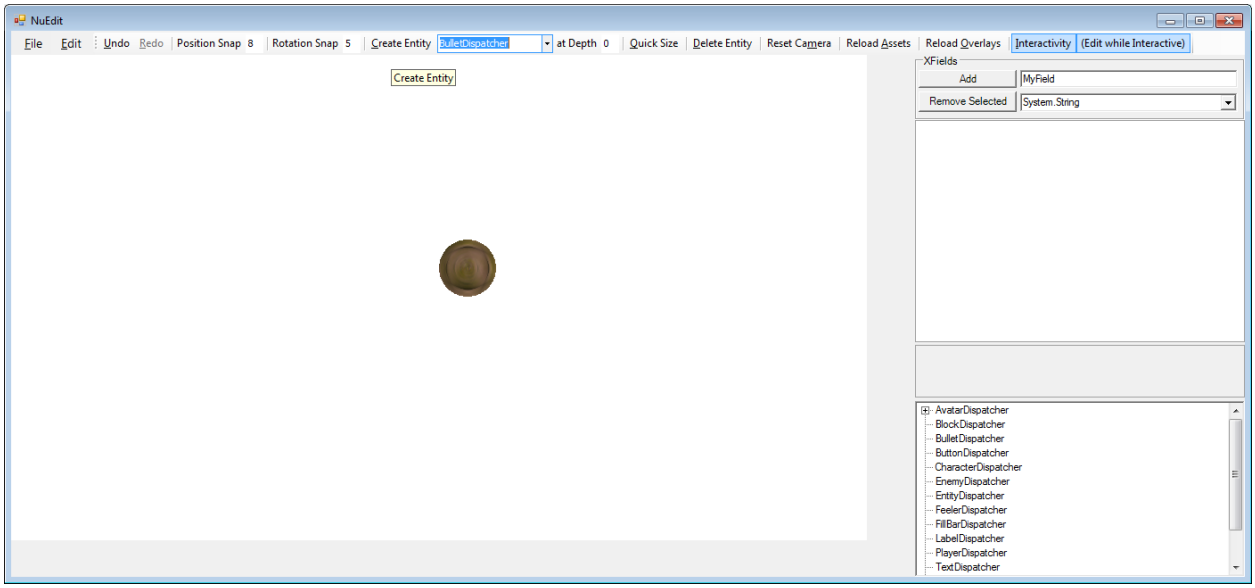


The bullet doesn't really have much behavior, but that's because the **Interactivity** button (top-left) is toggled off. Let's try toggling it on.

Whoops! It disappeared!

Don't worry. This is the defined behavior of a bullet in an interactive scene – it destroys itself after about half a second.

While keeping **Interactivity** on, let's see what happens to a bullet when it collides with something else. In the **Create Entity** drop-down menu, select the **AvatarDispatcher**, and then click **Create Entity**. You should end up with this –



Since the avatar is spawned right where the bullet is, the bullet should be destroyed due to collision as soon as it's created.

Select the **BulletDispatcher** again, and click **Create Entity** once more. You'll notice that a bullet is created and then instantly destroyed, perhaps pushing the avatar up just slightly.

By observing bullets in the editor, we can tell that their behavior is relatively simple – they render as a small blue dot, and are destroyed after a short period of time or after colliding with another entity.

The code behind the bullets

Now let's look at the BulletDispatcher implementation found in the BlazeDispatchers.fs file inside the BlazeVector project to understand how this behavior is implemented –

```
[<AutoOpen>]
module BulletDispatcherModule =

    type Entity with

        member bullet.Age = bullet?Age : int64
        static member setAge (value : int64) (bullet : Entity) = bullet?Age <- value

    type BulletDispatcher () =
        inherit EntityDispatcher ()

        let tickHandler event world =
            let (address, bullet : Entity, _) = Event.unwrap event
            if World.isGamePlaying world then
                let bullet = Entity.setAge (bullet.Age + 1L) bullet
                let world =
                    if bullet.Age < 28L then World.setEntity address bullet world
                    else snd <| World.removeEntity address bullet world
                (Propagate, world)
            else (Propagate, world)

        let collisionHandler event world =
            let (address, bullet : Entity, _) = Event.unwrap event
            if World.isGamePlaying world then
```

```
            let world = snd <| World.removeEntity address bullet world
            (Propagate, world)
        else (Propagate, world)

    static member FieldDefinitions =
        [define? Size <| Vector2 (24.0f, 24.0f)
         define? Density 0.25f
         define? Restitution 0.5f
         define? LinearDamping 0.0f
         define? GravityScale 0.0f
         define? IsBullet true
         define? CollisionExpression "Circle"
         define? SpriteImage PlayerBulletImage
         define? Age 0L]

    static member IntrinsicFacetNames =
        [typeof<RigidBodyFacet>.Name
         typeof<SpriteFacet>.Name]

    override dispatcher.Register (address, bullet, world) =
        let world = World.observe TickEventName address (CustomSub tickHandler) world
        let world = World.observe (CollisionEventName + address) address (CustomSub collisionHandler) world
        (bullet, world)
```

Let's break this code down piece by piece.

```
[<AutoOpen>]
module BulletDispatcherModule =
```

Dispatchers are defined in an auto-opened module with a matching name that is suffixed with the word **Module**. I personally believe all public types belong in auto-opened modules, so you will see such an approach taken consistently across the FPWorks repository.

```
type Entity with

    member entity.Age = entity?Age : int64
    static member setAge (value : int64) (entity : Entity) = entity?Age <- value
```

If you recall back to the **The Xtension System** section, you'll understand that a new field **Age** of type **int64** is being made accessible for entity types.

```
type [<Sealed>] BulletDispatcher () =
    inherit EntityDispatcher
```

Here begins the definition of the BulletDispatcher type. We notice that the BulletDispatcher type inherits from **EntityDispatcher** a dispatcher that provides no special capabilities.

```
    static member FieldDefinitions =
        [define? Size <| Vector2 (24.0f, 24.0f)
         define? Density 0.25f
         define? Restitution 0.5f
         define? LinearDamping 0.0f
         define? GravityScale 0.0f
         define? IsBullet true
         define? CollisionExpression "Circle"
         define? SpriteImage PlayerBulletImage
         define? Age 0L]
```

You might be wondering how dispatchers and facets specify what fields are added to entities. What you see before you is how! By specifying the static **FieldDefinitions** property for a given dispatcher or facet like so, the engine will automatically attach the defined fields with the corresponding values to the entity at run-time

The above definitions define the bullet's fields to give it physical and rendering properties becoming of a bullet, as well as initialize the user-defined **Age** field to 0.

```
        static member IntrinsicFacetNames =
            [typeof<RigidBodyFacet>.Name
             typeof<SpriteFacet>.Name]
```

By specifying the **intrinsicFacetNames** like so and exposing them via the static **IntrinsicFacetNames** property, the BulletDispatcher gains the following capabilities -

- Simple 2d physics body behavior and the corresponding fields
- Simple 2d sprite rendering behavior and the corresponding fields

The facets that are specified in this manner are known as "intrinsic facets". That is, their use is intrinsic to the definition of the entity to which the dispatcher is applied, and therefore they cannot be removed by altering the entity's **FieldNames** field.

The Register override

Next, we'll study the **Register** dispatch method override. Generally, the **Register** override defines what happens when an entity is added to the world. Conversely, there is an **Unregister** method that defines what happens when the entity is removed from the world (though an override for Unregister is not used here).

Here we see what **Register** does in the BulletDispatcher -

```
        override dispatcher.Register (address, entity, world) =
            let (entity, world) = base.Register (address, entity, world)
            let world = World.observe TickEventName address (CustomSub tickHandler) world
            let world = World.observe (CollisionEventName + address) address (CustomSub
collisionHandler) world
            (entity, world)
```

The first line calls the **base.Register** method for obvious reasons. The second and third lines are used to observe and react to certain events for the duration of the entity's lifetime. The dispatcher's **tickHandler** function will be called whenever the tick event is published (see the previous section **The Purely-Functional Event System**), and the dispatcher's **collisionHandler** will be called whenever a collision takes place on the bullet.

The tickHandler

Here's the code used to define the behavior of **tickHandler** –

```
        let tickHandler event world =
            let (address, bullet : Entity, _) = Event.unwrap event
            if World.isGamePlaying world then
                let bullet = Entity.setAge (bullet.Age + 1L) bullet
                let world =
                    if bullet.Age < 28L then World.setEntity address bullet world
                    else snd <| World.removeEntity address bullet world
                (Propagate, world)
            else (Propagate, world)
```

Its work is simple, but the idioms may be new.

First note the **Event.unwrap** call. It is a helper function that takes two type parameters and an event value. In typical usage like here, it infers its type parameters from its return target. It returns a triple containing 1) the current simulant's address, 2) the simulant itself casted to the first type parameter (here, Entity), and 3) the

event's data casted as the second type parameter (here, obj since its unused). This function may look weird, but it saves you from doing a bunch of transformations on the entity data just to get the most relevant data out of it.

The next line exists to attain different behavior depending on whether the editor is in **Interactive** mode. Here the tickHandler checks to see if the game is actually playing before doing anything. If the game is playing, the tickHandler increments the bullet's **Age** field, checks if the bullet is older than 28 ticks, and then destroys it if so.

The collisionHandler

Here's the code used to define the **collisionHandler** –

```
let collisionHandler event world =
    let (address, bullet : Entity, _) = Event.unwrap event
    if World.isGamePlaying world then
        let world = snd <| World.removeEntity address bullet world
        (Propagate, world)
    else (Propagate, world)
```

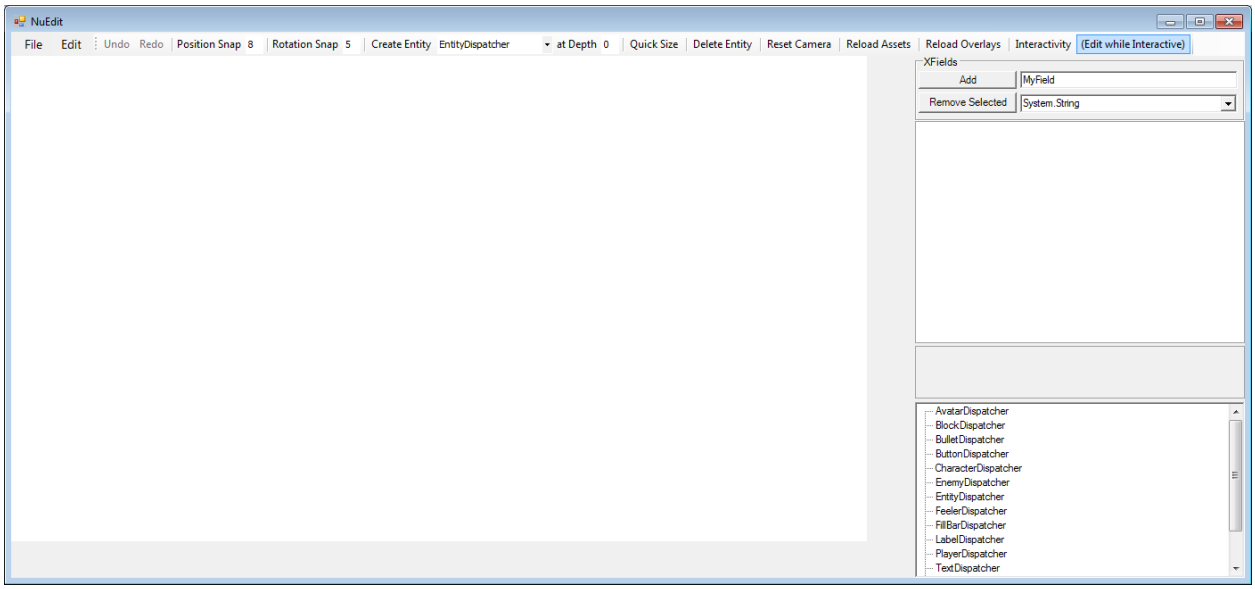
Even simpler than the previous handler, it simply destroys the bullet when a collision with it takes place while the game is playing.

Enemies and the EnemyDispatcher

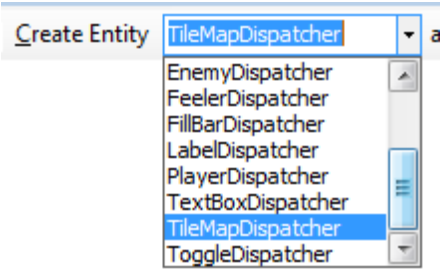
Since we have bullets, we obviously need something to shoot them at! In BlazeVector, we use little Army-men style bad guys that charge across the screen.

Enemies in NuEdit

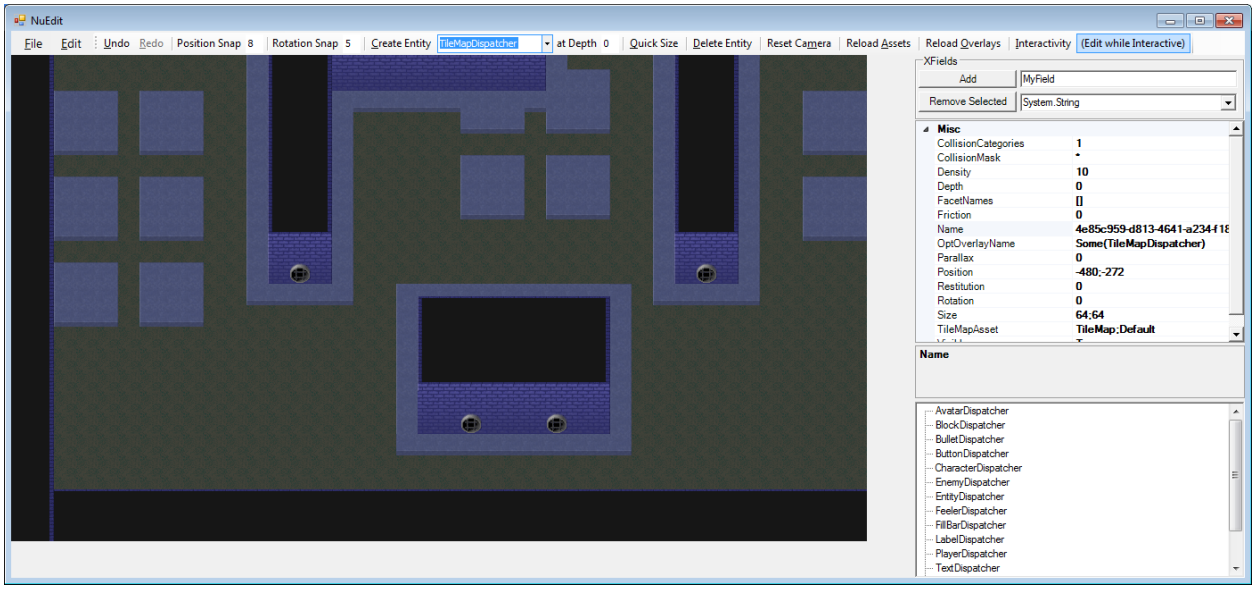
First, let's open **NuEdit** like before and again select **BlazeVector.exe** as the game's execution file. The editor will be opened as normal -



Before creating enemies, let's create a tile map in which for them to live by selecting the **TileMapDispatcher**, clicking **Create Entity**, and then dragging the bottom-left corner of the tile map to the bottom left corner of the screen -

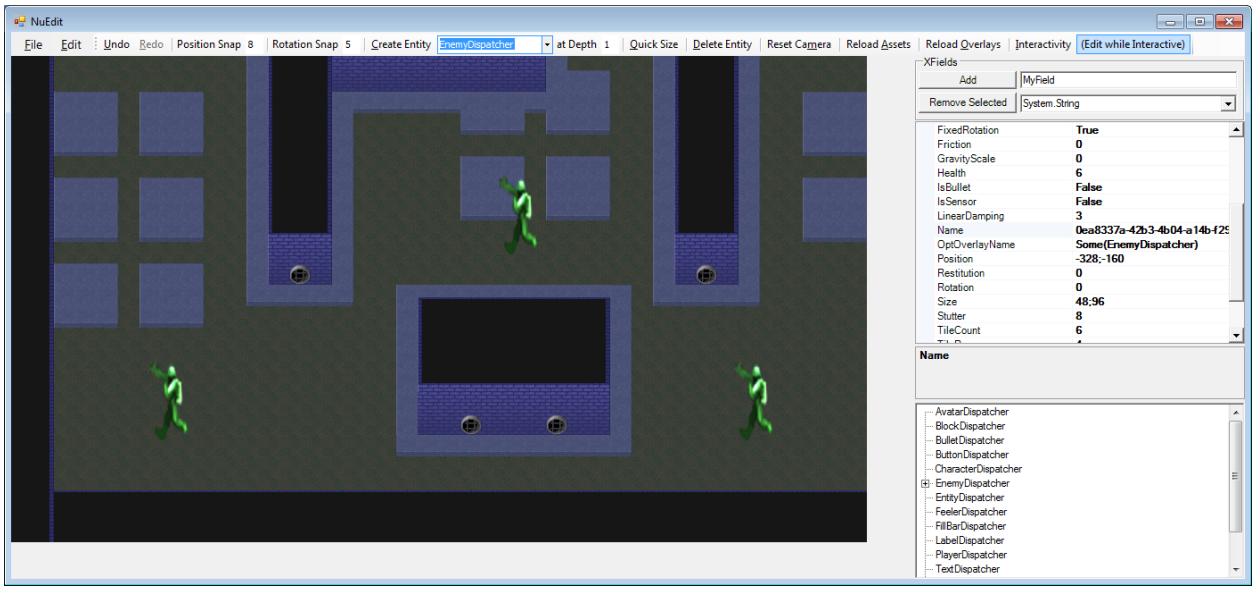


We will end up with something like this –

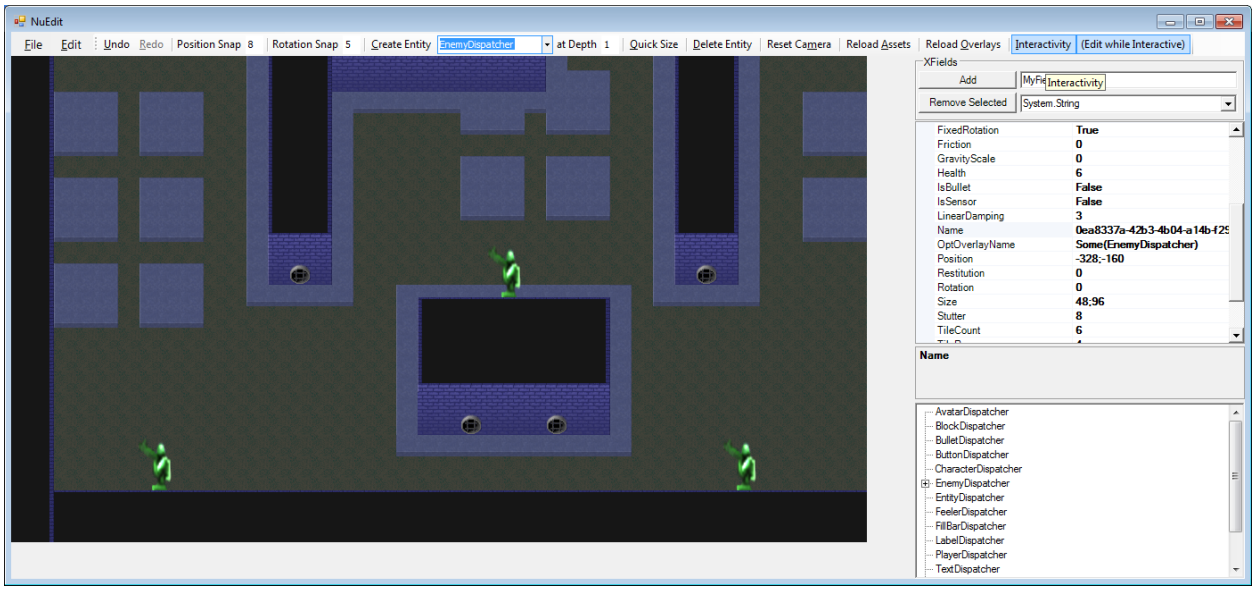


Since we want to create additional entities on top of the bottom layer of the tile map (but below the second layer), we set **at Depth** value in the editor's tool bar to **1**. Next, we create a few enemies by selecting the **Enemy Dispatcher** and clicking **Create Entity** a few times (or alternatively, by right-clicking a few different spots on the tile map and pressing **R**). Once the enemies are created, move them around to different positions on the map.

We might end up with something like this –



Now, for fun, let's toggle the **Interactivity** button to see what these enemies would do during gameplay.



They just drop to the ground and walk to the left!

To revert to our previous state before enabling **Interactivity**, press the **Undo** button. This will disable **Interactivity** and put the enemies back where they started.

TODO: more details on BlazeVector's implementation.

More Engine Details

Assets and the AssetGraph

Nu has a special system for efficiently and conveniently handling assets called the **Asset Graph**. The Asset Graph is configured in whole by an XML file named **AssetGraph.xml**. This file is included in every new Nu game project, and is placed in the same folder as the project's **Program.fs** file.

The first thing you might notice about assets in Nu is that they are not built like normal assets via Visual Studio. The Visual Studio projects themselves need to have no knowledge of a game's assets. Instead, assets are built by a program called **NuPipe.exe**. NuPipe knows what assets to build by itself consulting the game's Asset Graph. During the build process of a given Nu game project, NuPipe is invoked from the build command line like so –

"$(ProjectDir)..\..\..\Nu\Nu\NuPipe\bin\$(ConfigurationName)\NuPipe.exe" "$(ProjectDir)\" "$(TargetDir)\"

NuPipe references the game's Asset Graph to automatically copy all its asset files to the appropriate output directory. Note that for speed, NuPipe copies only missing or recently altered assets.

Let's study the structure of the data found inside the AssetGraph.xml file that ultimately defines a game's Asset Graph –

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <Package name="Default">
    <Asset name="Image" fileName="Assets/Default/Image.png" associations="[Rendering]"/>
    <Asset name="Image2" fileName="Assets/Default/Image2.png" associations="[Rendering]"/>
    <Asset name="Image3" fileName="Assets/Default/Image3.png" associations="[Rendering]"/>
    <Asset name="Image4" fileName="Assets/Default/Image4.png" associations="[Rendering]"/>
    <Asset name="Image5" fileName="Assets/Default/Image5.png" associations="[Rendering]"/>
    <Asset name="Image6" fileName="Assets/Default/Image6.png" associations="[Rendering]"/>
    <Asset name="Image7" fileName="Assets/Default/Image7.png" associations="[Rendering]"/>
    <Asset name="Image8" fileName="Assets/Default/Image8.png" associations="[Rendering]"/>
    <Asset name="Image9" fileName="Assets/Default/Image9.png" associations="[Rendering]"/>
    <Asset name="Image10" fileName="Assets/Default/Image10.png" associations="[Rendering]"/>
    <Asset name="Font" fileName="Assets/Default/FreeMonoBold.032.ttf" associations="[Rendering]"/>
    <Asset name="TileSet" fileName="Assets/Default/TileSet.png" associations="[Rendering]"/>
    <Asset name="Sound" fileName="Assets/Default/Sound.wav" associations="[Audio]"/>
    <Asset name="Song" fileName="Assets/Default/Song.ogg" associations="[Audio]"/>
    <Asset name="TileMap" fileName="Assets/Default/TileMap.tmx" associations="[]"/>
  </Package>
</Root>
```

You'll notice inside the **Root** node that there is a single **Package** node that in turn holds multiple **Asset** nodes. In Nu, a single asset will never be loaded by itself. Instead, a package of assets containing the desired asset is loaded (or unloaded) all at once. The Asset Graph allows you to conveniently group together related assets in a package so they can be loaded as a unit.

Further, the use of the Asset Graph allows (well, *forces*) you to refer to assets by their asset and package name rather than their raw file name. Instead of setting a sprite image field to **Assets/Default/Image.png** (which absolutely will not work), you must instead set it to **Image;Default** (assuming you want to load it from the **Default** package).

You may notice that there is no need to manually specify which assets will be loaded in your game before using them. This is because when an asset is used by the rendering or audio system, it will automatically have its associated package loaded on-demand. This is convenient and works great in NuEdit, but this is not always what

you want during gameplay. For example, if the use of an asset triggers a package load in the middle of an action sequence, the game could very well stall during the IO operations, thus resulting in an undesirable pause. Whenever this happens, a note will be issued to the console that an asset package was loaded on-the-fly. Consider this a performance warning for your game.

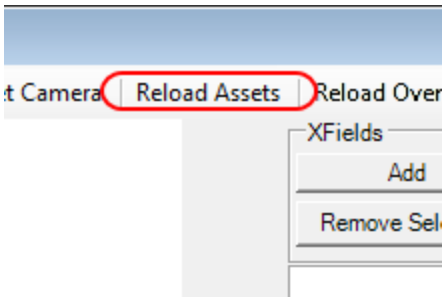
Fortunately, there is a simple way to alleviate the potential issue. When you know that the next section of your game will require a package of rendering assets, you can send a 'package use hint' to the renderer like so –

```
let world = World.hintRenderingPackageUse "MyPackageName" world
```

Currently, this will cause the renderer to immediately load all the all the assets in the package named **MyPackageName** which are associated with the rendering system (which assets are associated with which system(s) is specified by the **associations** attribute of the Asset node in AssetGraph.xml). Notice that this message is just a hint to the renderer, not an overt command. A future renderer may have different underlying behavior such as using asset streaming.

Conversely, when you know that the assets in a loaded package won't be used for a while, you can send a 'package disuse hint' to unload them via the corresponding **World.hintRenderingPackageDisuse** function.

Finally, there is a nifty feature in NuEdit where the game's currently loaded assets can be rebuilt and reloaded at run-time. This is done by pressing the **Reload Assets** button that is highlighted here –



Serialization and Overlays

By default, all of your simulation types can be serialized at any time to an XML file. No extra work will generally be required on your behalf to make serialization work, even when making your own custom dispatchers.

However, there are limitations. For example, because F# doesn't come with a way to serialize its discriminated unions, any field that has a DU as its type will not be serialized. In these instances, you will need to create and register an appropriate type provider. Nu has a few out of the box, once of which will be shown as an example here –

```
    /// The type of a physics body; Static, Kinematic, or Dynamic.
    type [<StructuralEquality; NoComparison; TypeConverter (typeof<BodyTypeTypeConverter>)>] BodyType =
        | Static
        | Kinematic
        | Dynamic

    /// Converts BodyType types.
    /// TODO: factor out a simple DU type converter.
    and BodyTypeTypeConverter () =
        inherit TypeConverter ()
        override this.CanConvertTo (_, destType) =
            destType = typeof<string>
        override this.ConvertTo (_, _, source, _) =
            let bodyType = source :?> BodyType
```

```
            match bodyType with
            | Static -> "Static" :> obj
            | Kinematic -> "Kinematic" :> obj
            | Dynamic -> "Dynamic" :> obj
        override this.CanConvertFrom (_, sourceType) =
            sourceType = typeof<Vector2> || sourceType = typeof<string>
        override this.ConvertFrom (_, _, source) =
            let sourceType = source.GetType ()
            if sourceType = typeof<BodyType> then source
            else
                match source :?> string with
                | "Static" -> Static :> obj
                | "Kinematic" -> Kinematic :> obj
                | "Dynamic" -> Dynamic :> obj
                | other -> failwith <| "Unknown BodyType '" + other + "'."
```

To manually stop any given field from being serialized, simply end its name with the letter 'Np' (that's capital 'N', lowercase 'p' – stands for non-persistent). Additionally, fields that end with 'Id', or 'Ids', will not be serialized.

Additionally, when you save a scene in NuEdit, you may notice that not all of a given entity's fields at actually written out, even if they don't end with 'Np' or the like. This is a good thing, and is our next feature in action – **Overlays**.

Overlays accomplish two extremely important functions in Nu. First, they reduce the amount of stuff written out to (and consequently read in from) serialization files. Second, they provide the user with a way to abstract over field values that multiple entities hold in common. Overlays are defined in a file that is included with every new Nu game project called **Overlay.xml**. Additionally, for every dispatcher and facet type that the engine is informed of, an overlay with a matching name is defined with values set to the type's **FieldDefinitions**.

Let's take a look at the definition of some overlays now –

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>

  <!-- Add user-defined facet overlays here... -->

  <SampleFacetOverlay>
    <BodyType>Dynamic</BodyType>
    <Friction>0.5</Friction>
    <CollisionExpression>Circle</CollisionExpression>
  </SampleFacetOverlay>

  <!-- Add user-defined dispatcher overlays here... -->

  <SampleDispatcherOverlay include="[EntityDispatcher]">
    <Size>100;100</Size>
    <FacetNames>[SpriteFacet]</FacetNames>
  </SampleDispatcherOverlay>

</Root>
```

Where overlays get interesting is when they are applied to an entity at run-time. Say you're in NuEdit and you want to have a common set of button field values to which you can apply to multiple buttons. Since we're talking UI, let's refer to this as a 'style'. Say also that this new button style is defined to have a custom click sound and to be disabled by default. Instead of manually setting each of these fields on each button, you can create an overlay that describes the style and then apply that overlay to the desired buttons. The steps are described as such -

First, add an overlay like this to your Overlay.xml file (ensuring the specified click sound asset exists and is also declared in the AssetGraph.xml file) –

```
<MyButtonOverlay include="[ButtonDispatcher]">
  <Enabled>False</Enabled>
  <ClickSound>MyClickSound;MyUiPackage</ClickSound>
</MyButtonOverlay>
```

Second, click the **Reload Overlays** button near the top-right of the editor.

Third, for each button you wish to overlay, change its OptOverlayName field to **Some(MyButtonOverlay)**.

Voila! Both the **Enabled** and **ClickSound** fields will be changed to correspond to the values specified in the new overlay on each button.

Well, that is if you've NOT changed the value of the fields to something other than what was specified in its previous overlay! You see, overlay values are applied to only to the fields which haven't been changed from the current overlay's values. In this manner, overlays can act as a styling mechanism while still allowing you to customize the overlaid fields post hoc.

Finally, overlays have a sort of 'multiple inheritance' where one overlay can include all the overlay values of one or more other overlays recursively. This is done by specifying the **include** attribute in the Overlay.xml file (like is done with **include="[ButtonDispatcher]"** above).

Taken together, overlays avoid a ton of duplication while allowing changes to them to automatically propagate to the entities to which they are applied.

Subsystems and Message Queues

Fortunately, Nu is not a monolithic game engine. The definition of its simulation types and the implementation of the subsystems that process / render / play them are separate. They are so separate, in fact, that neither the engine, nor the dispatchers that define the behavior of simulation types, are allowed to send commands to the subsystems directly (note I said 'commands', the engine does send non-mutating queries the subsystems directly, though user code should not even do this). Instead of sending commands directly, each subsystem must be interfaced with via its own respective message queue.

Thankfully, there are convenience functions on the World type that make this easy. Remember the **World.hintRenderingPackageUse** function? That is one of these convenience functions, and all of them are as easy to use. However, accessing additional functionality from any of the subsystems will require writing new messages for them, in turn requiring a change to the engine code. Fortunately, there is an easy way to enable creating new types of messages without requiring changes to the engine, and that will be implemented shortly (if it hasn't already by the time you read this).

Now, of course the use of message queues can make accomplishing certain things a little more complicated due to the inherent indirection it entails. Not only is the call-site a bit separated from the target, the time at which the actual message is handled is also separated. These two facts can make debugging a little more challenging. What does this indirection buy us that such additional difficulty is warranted?

For one, you'll notice that the API presented by each of the subsystems is inherently impure / stateful. If either the engine or user code were to invoke these APIs directly, the functional purity of both would be compromised,

and all the nice properties that come from it destroyed. And secondly, it is likely that one or more of the subsystems will eventually be put on a thread separate from the game engine anyway, thus making the message queues unavoidable anyhow.

Currently, the subsystems used in Nu include a **Rendering** subsystem, an **Audio** subsystem, and a **Physics** subsystem. Additional subsystems such as **Artificial Intelligence**, **Particles**, and et al can be added by modifying the engine directly. Perhaps an extension point could be added to the engine such that adding subsystems would not require modifying engine code, but that's a task that's not yet been investigated.