Nu Game Engine

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

Bryan Edds, 2014

Table of Contents

What's It All About?

The Nu Game Engine is a **Simple**, **Purely-Functional**, **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

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. There are some imperative operations going on in Nu behind the scenes for speed. 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, 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.

Please note that I intend to, at some point, implement 3d capabilities in Nu. Nu was designed such that the addition of 3d functionality is not precluded. Unfortunately, due to a lack of resources to fund such an implementation, there is currently no timeframe for this.

F#

We know what F# is, so why use it? First, because of its **cross-platform** nature. 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 Scala, F# offers a simple and easy-to-use programming model. Unlike JVM languages generally, 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 that 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 the engine or step debug into it!

The next thing you must do is to ensure you have the **vcredist_x86** and / or **vcredist_x64** redistributables installed in your development environment. This is due to Nu's dependency on **Magick.NET** and its dependency on one or both of these redistributables. See this Stack Overflow post to get the relevant links and details - http://stackoverflow.com/questions/23308415/could-not-load-file-or-assembly-magick-net-x86-dll-or-one-of-its-dependencies

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 Nu solution, first make sure to have **Visual Studio 2013** installed (the free **Express** version is fine). 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 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 Nu 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 –
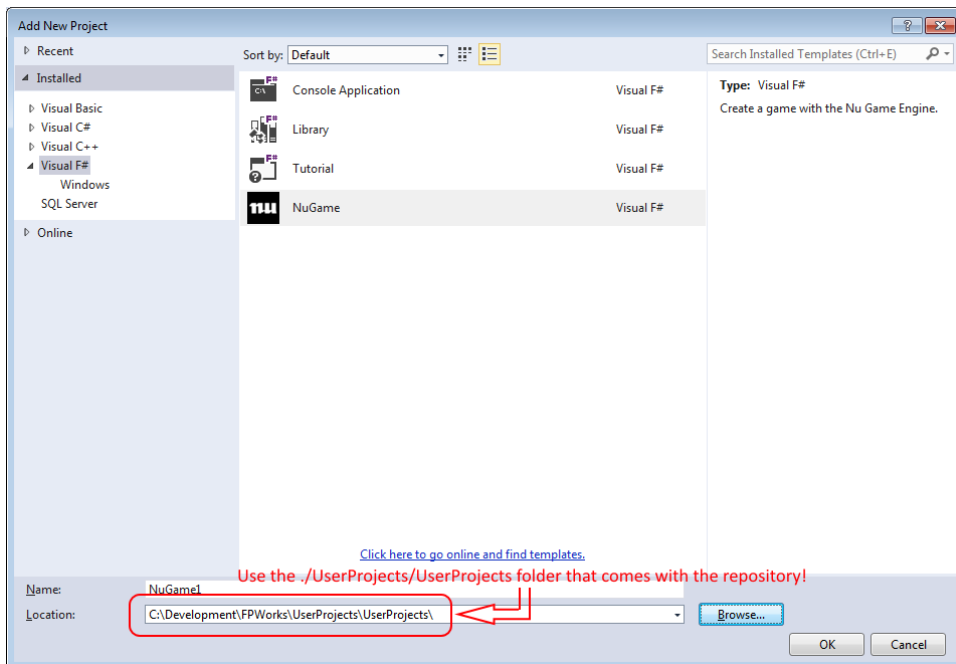
If this is done incorrectly, the new project will not be able to find the Nu, NuPipe , Prime, and SDL2# dependencies needed to build it!

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.

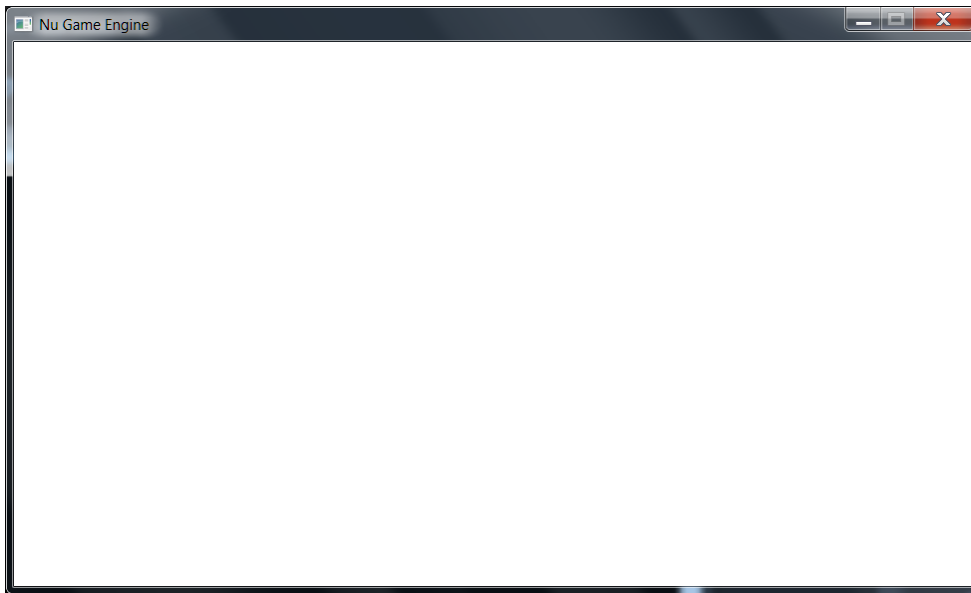When the new project is run from Visual Studio, you'll notice a window popping up that is filled with a pure white color. By default, Nu does nothing but clear the frame buffer with white pixels. There is no interactivity in your program, as the engine is not yet being told to do anything.

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 System
open FSharpx
open SDL2
open OpenTK
open Prime
open Nu
open Nu.Constants
open Nu.WorldConstants
open Nu.Observation
open Nu.Chain

// this is the main module for our program.
module Program =

    // this is a plugin for the Nu game engine by which user-defined dispatchers, facets, and other
    // sorts of values can be obtained by both your application and NuEdit. Currently, there are no
    // overrides for its factory methods since there are no user-defined dispatchers, facets, et al
    // defined for this project yet.
    type NuGame1Plugin () =
        inherit NuPlugin ()

    // this the entry point for the your Nu application
    let [<EntryPoint; STAThread>] 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 =

            // an instance of the above plugin
            let nuPlugin = NuGame1Plugin ()

            // here is an attempt to make the world with the various initial states, the engine
            // plugin, and SDL dependencies. Note that the first boolean, when true, protects from
            // a Farseer physics bug but slows down the engine badly, so that's why it's false.
            World.tryMake false true 1L () nuPlugin sdlDeps

        // this is a callback that specifies your game's unique behavior when updating the world
        // every frame. The World value is the state of the world after the callback transforms
        // 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 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 **NuPlugin** type, 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 in 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.
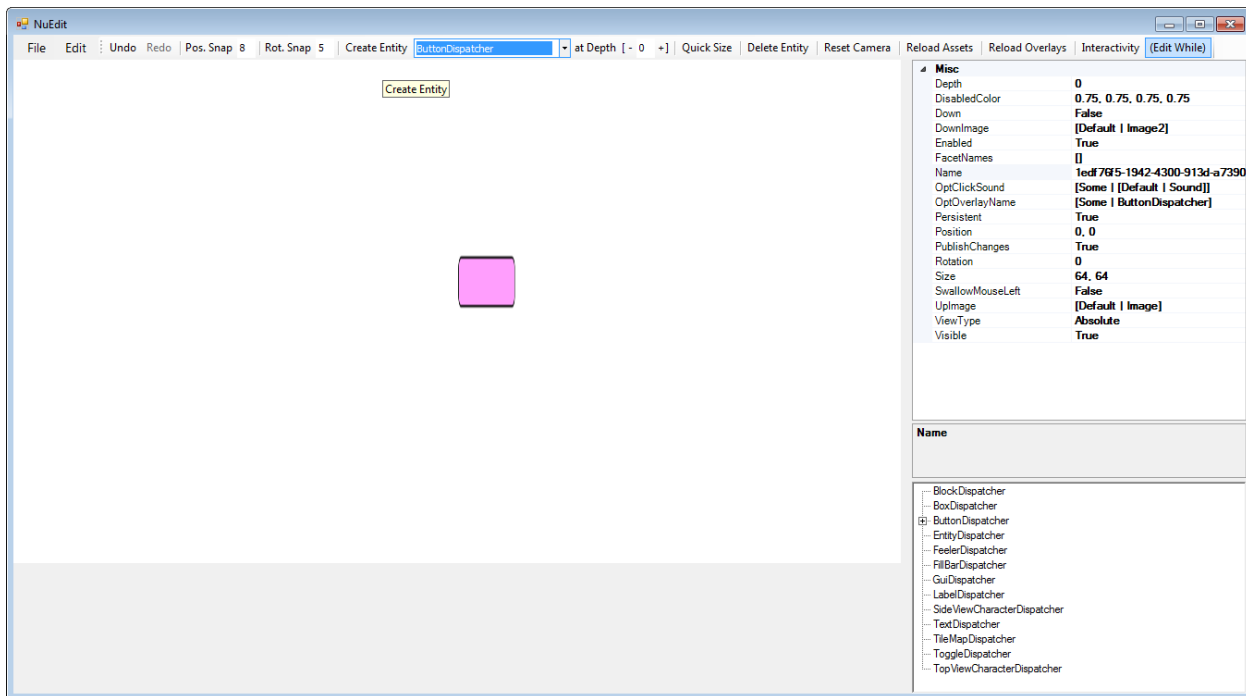
You'll notice a squished button appear in the middle of the editing panel. By default, most entities are created with a size of (64, 64). Fortunately, Nu gives you an easy way to resize the entity to fit the button's image by pressing the **Quick Size** button. Press it now.
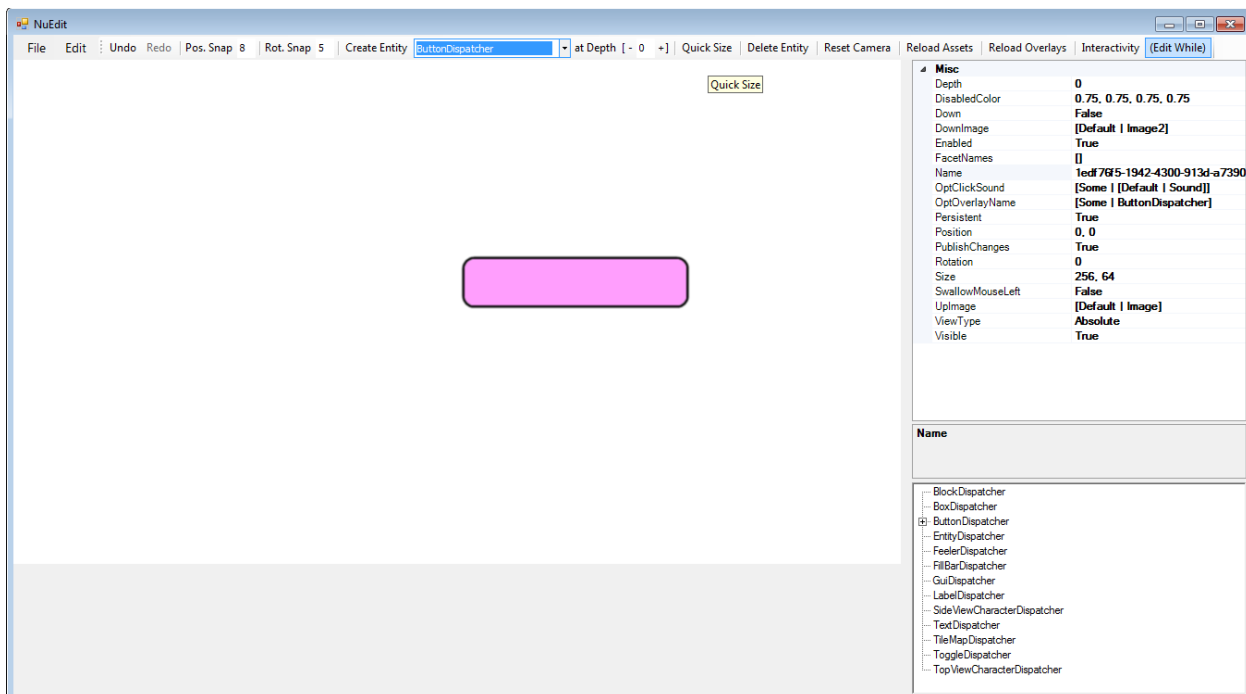
We have a full-sized button! Now notice that the property grid on the right has been populated with the button's fields. These fields can be edited to change the button however you like. For a button that will be used to control the game's stat, the first thing you will want to do is to give it an appropriate 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 –

Let's rename the tile map to **MyTileMap**. Then, let us click and drag the tile map so its bottom-left corner lines up with the bottom 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!
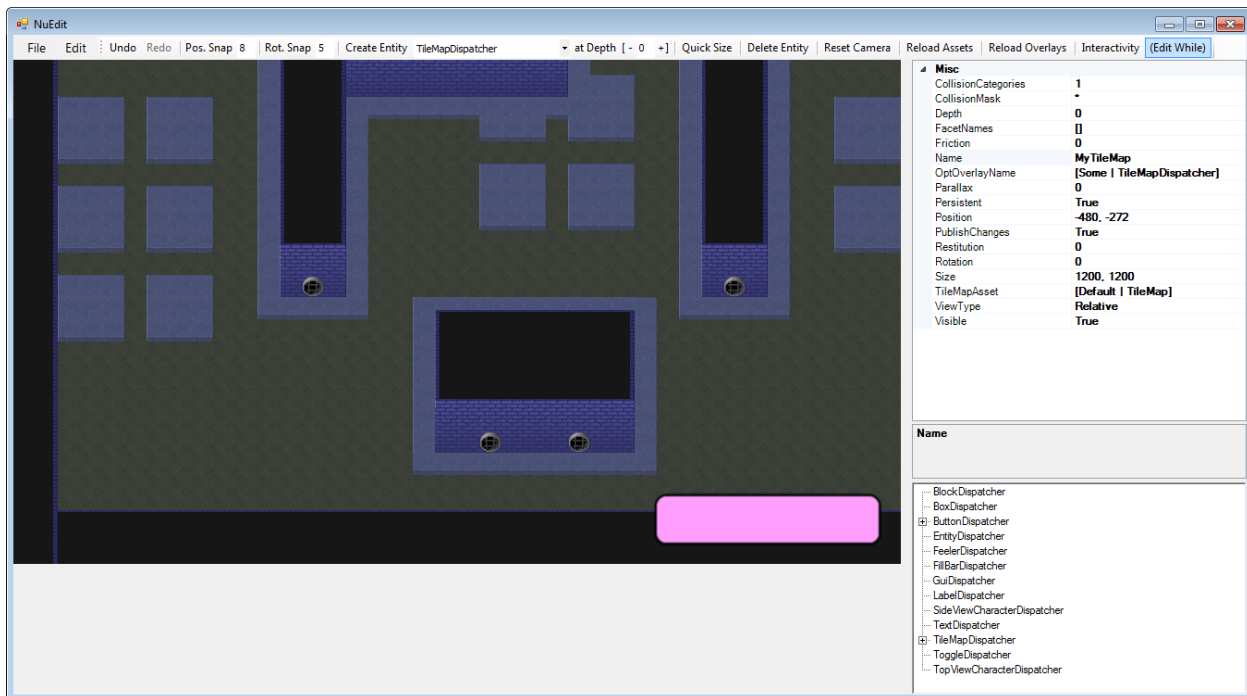
Now click and drag with the MIDDLE mouse button to change the position of the camera that is used to view the game. Check out your lovely new tile map! If your camera gets lost in space, click the **Reset Camera** button that is to the left of the **Interact** button.

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 a **1** or click the **+** button to its right. 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...**

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 –

```fsharp
namespace BlazeVector
open Nu
open Nu.Constants
open Nu.WorldConstants
module BlazeConstants =

    // these constants specify the packages as named in the project's 'AssetGraph.xml' file
    let GuiPackageName = "Gui"
    let GameplayPackageName = "Gameplay"

    // these constants specify the file paths from which various simulants are loaded
    let TitleGroupFilePath = "Assets/Gui/Title.nugroup"
    let CreditsGroupFilePath = "Assets/Gui/Credits.nugroup"
    let HudGroupFilePath = "Assets/Gui/Hud.nugroup"
    let PlayerGroupFilePath = "Assets/Gameplay/Player.nugroup"
    let Section0FilePath = "Assets/Gameplay/Section0.nugroup"
    let Section1FilePath = "Assets/Gameplay/Section1.nugroup"
```

```
let Section2FilePath = "Assets/Gameplay/Section2.nugroup"
let Section3FilePath = "Assets/Gameplay/Section3.nugroup"
let SectionFilePaths = [Section0FilePath; Section1FilePath; Section2FilePath; Section3FilePath]

// these constants locate various assets described by the project's 'AssetGraph.xml' file
let NuSplashSound = { PackageName = GuiPackageName; AssetName = "Nu" }
let MachinerySong = { PackageName = GuiPackageName; AssetName = "Machinery" }
let DeadBlazeSong = { PackageName = GameplayPackageName; AssetName = "DeadBlaze" }
let HitSound = { PackageName = GameplayPackageName; AssetName = "Hit" }
let ExplosionSound = { PackageName = GameplayPackageName; AssetName = "Explosion" }
let ShotSound = { PackageName = GameplayPackageName; AssetName = "Shot" }
let JumpSound = { PackageName = GameplayPackageName; AssetName = "Jump" }
let DeathSound = { PackageName = GameplayPackageName; AssetName = "Death" }
let EnemyBulletImage = { PackageName = GameplayPackageName; AssetName = "EnemyBullet" }
let PlayerBulletImage = { PackageName = GameplayPackageName; AssetName = "PlayerBullet" }
let EnemyImage = { PackageName = GameplayPackageName; AssetName = "Enemy" }
let PlayerImage = { PackageName = GameplayPackageName; AssetName = "Player" }

// this constant describes the 'dissolving' transition behavior of game's screens
let DissolveData =
    { IncomingTime = 20L
      OutgoingTime = 30L
      DissolveImage = DefaultDissolveImage }

// these pair of constants are -
//   a) a string used to give a name to the splash screen
//   b) a proxy used to locate and operate upon the splash screen
// A screen proxy is created by first converting a name to an address with the ntoa function, and
// then passing the address into the Screen.proxy function.
let SplashName = "Splash"
let Splash = Screen.proxy <| ntoa SplashName

// this constant describes the 'splashing' behavior of game's splash screen
let SplashData =
    { DissolveData = DissolveData
      IdlingTime = 60L
      SplashImage = { PackageName = DefaultPackageName; AssetName = "Image5" }}

// these are like the pair of constants for splash screen above, but for the title screen
let TitleName = "Title"
let Title = Screen.proxy <| ntoa TitleName

// these are also like the pair of constants above, but for the group that is loaded into the
// title that contains all of its gui entities. You'll notice that the group is proxied from a
// combination of the address of its containing screen as well as its own personal name as
// found in its originating document, 'Assets/Gui/Title.nugroup'.
//
// You'll need to familiarize yourself with the 'satoga' operator and its relatives by reading
// their respective documentation comments.
let TitleGroupName = "Group"
let TitleGroup = Group.proxy <| satoga Title.ScreenAddress TitleGroupName

// these are like the above, but for the play button found in the above group
let TitlePlayName = "Play"
let TitlePlay = Entity.proxy <| gatoea TitleGroup.GroupAddress TitlePlayName

// and so on...
let TitleCreditsName = "Credits"
let TitleCredits = Entity.proxy <| gatoea TitleGroup.GroupAddress TitleCreditsName

// and so on.
let TitleExitName = "Exit"
let TitleExit = Entity.proxy <| gatoea TitleGroup.GroupAddress TitleExitName

// these constants specify names and proxies for various simulants of the gameplay screen
let GameplayName = "Gameplay"
let Gameplay = Screen.proxy <| ntoa GameplayName
let GameplayGroupName = "Group"
let GameplayGroup = Group.proxy <| satoga Gameplay.ScreenAddress GameplayGroupName
let GameplayBackName = "Back"
let GameplayBack = Entity.proxy <| gatoea GameplayGroup.GroupAddress GameplayBackName
let PlayerGroupName = "Player"
let PlayerGroup = Group.proxy <| satoga Gameplay.ScreenAddress PlayerGroupName
let PlayerName = "Player"
let Player = Entity.proxy <| gatoea PlayerGroup.GroupAddress PlayerName
```

```
    // these constants specify names and proxies for various simulants of the credits screen
    let CreditsName = "Credits"
    let Credits = Screen.proxy <| ntoa CreditsName
    let CreditsGroupName = "Group"
    let CreditsGroup = Group.proxy <| satoga Credits.ScreenAddress CreditsGroupName
    let CreditsBackName = "Back"
    let CreditsBack = Entity.proxy <| gatoea CreditsGroup.GroupAddress CreditsBackName

    // and finally, this constant simply specifies how many sections are added to a game
    let SectionCount = 32
```

Hopefully the comments in the document will give you sufficient detail for now. Let us turn to the code in the **Program.fs** file –

```
namespace BlazeVector
open System
open OpenTK
open SDL2
open Prime
open Nu
open Nu.Constants
open Nu.WorldConstants
open BlazeVector
module Program =

    /// Creates BlazeVector-specific values (here dispatchers and facets).
    /// Allows BlazeVector simulation types to be created in the game as well as in NuEdit.
    type BlazePlugin () =
        inherit NuPlugin ()

        // make our game-specific game dispatcher...
        override this.MakeOptGameDispatcher () =
            Some (BlazeDispatcher () :> GameDispatcher)

        // make our game-specific screen dispatchers...
        override this.MakeScreenDispatchers () =
            [GameplayScreenDispatcher () :> ScreenDispatcher]

        // make our game-specific group dispatchers...
        override this.MakeGroupDispatchers () =
            [PlayerGroupDispatcher () :> GroupDispatcher]

        // make our game-specific entity dispatchers...
        override this.MakeEntityDispatchers () =
            [BulletDispatcher () :> EntityDispatcher
             PlayerDispatcher () :> EntityDispatcher
             EnemyDispatcher () :> EntityDispatcher]

    // this the entry point for the BlazeVector application
    let [<EntryPoint; STAThread>] 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 }

                    // 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 =

                        // an instance of the above plugin
                        let nuPlugin = BlazePlugin ()

                        // here is an attempt to make the world with the various initial states, the engine
                        // plugin, and SDL dependencies. Note that the first boolean, when true, protects from
                        // a Farseer physics bug but slows down the engine badly, so that's why it's false.
                        World.tryMake false true 1L () nuPlugin sdlDeps

                    // this is a callback that specifies your game's unique behavior when updating the world
                    // every update. The World value is the state of the world after the callback transforms
                    // 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 the game. We're off and running!
                    World.run tryMakeWorld updateWorld sdlConfig
```

We've seen most of this before aside from overrides being provided to the engine plugin, the window being titled "BlazeVector", and the world creation actually happens in the **Register** callback of **BlazeDispatcher**. Let's investigate into **BlazeDispatcher.fs** to learn a little more about it –

```
namespace BlazeVector
open OpenTK
open SDL2
open Prime
open Nu
open Nu.Constants
open Nu.WorldConstants
open BlazeVector
open BlazeVector.BlazeConstants

[<AutoOpen>]
module BlazeDispatcherModule =

    /// The custom type for BlazeVector's game dispatcher.
    type BlazeDispatcher () =
        inherit GameDispatcher ()

        // this function handles the selection of the title screen by playing the song "Machinery"
        static let handleSelectTitleScreen _ world =
            let world = World.playSong 0 1.0f MachinerySong world
            (Cascade, world)

        // this function handles the clicking of the play button on the title screen by playing
        // the game
        static let handleClickTitlePlay _ world =
            let world = World.fadeOutSong DefaultTimeToFadeOutSongMs world
            let world = World.transitionScreen Gameplay world
            (Cascade, world)

        // this function creates the BlazeVector title screen to the world
        static let createTitleScreen world =

            // this creates a dissolve screen from the specified file with the given parameters
            let world = snd <| World.createDissolveScreenFromGroupFile false DissolveData typeof<ScreenDispatcher>.Name
TitleGroupFilePath (Some TitleName) 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".
            //
            // You will need to familiarize yourself with the calling conventions of the various
            // World.subscribe functions as well as the event address operators '->>-' and its ilk
            // by studying their types and documentation comments.
            let world = World.subscribe4 handleSelectTitleScreen (SelectEventAddress ->>- Title.ScreenAddress) Game 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 Gameplay screen
        let world = World.subscribe4 handleClickTitlePlay (ClickEventAddress ->>- TitlePlay.EntityAddress) Game 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 (World.handleAsScreenTransition Credits) (ClickEventAddress ->>-
TitleCredits.EntityAddress) Game world

        // subscribes 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 World.handleAsExit (ClickEventAddress ->>- TitleExit.EntityAddress) Game world

    // pretty much the same as above, but for the Credits screen
    static let createCreditsScreen world =
        let world = snd <| World.createDissolveScreenFromGroupFile false DissolveData typeof<ScreenDispatcher>.Name
CreditsGroupFilePath (Some CreditsName) world
        World.subscribe4 (World.handleAsScreenTransition Title) (ClickEventAddress ->>- CreditsBack.EntityAddress) Game world

    // and so on.
    static let createGameplayScreen world =
        let world = snd <| World.createDissolveScreenFromGroupFile false DissolveData typeof<GameplayScreenDispatcher>.Name
HudGroupFilePath (Some GameplayName) world
        World.subscribe4 (World.handleAsScreenTransition Title) (ClickEventAddress ->>- GameplayBack.EntityAddress) Game world

    // game registration is where the game's high-level logic is set up!
    override dispatcher.Register _ world =

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

        // create our screens
        let world = createTitleScreen world
        let world = createCreditsScreen world
        let world = createGameplayScreen world

        // create a splash screen that automatically transitions to the Title screen
        let (splash, world) = World.createSplashScreen false SplashData typeof<ScreenDispatcher>.Name Title (Some SplashName)
world

        // play a neat sound effect, select the splash screen, and we're off!
        let world = World.playSound 1.0f NuSplashSound world
        World.selectScreen splash world
```

* Apologies for the small font; it can be difficult to make code fit on a portrait document layout!

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.

One thing that may seem strange is the use of the term 'proxy' in the above code. This deserves discussion. In Nu, one does not transform simulant values directly, but rather through proxies into the world. This is done to prevent the presence of local copies of simulant values which can be invalidated when the world is transformed. This is a fine point, perhaps, and also one that you can likely ignore for now so long as you stick to the API provided by Nu.

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 Gui (*graphical user interface*) elements are an intrinsic part of games. Rather than tacking on a Gui system like other engines, Nu implements its Gui components directly as entities. There is no arbitrary divide between a say a box with physics and a Gui 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 contains the simulation values starting with the Game. In addition to that, it contains facilities needed to execute a game such as various subsystems (such as a render context, an audio context, physics, and those defined by the user), a purely-functional message system (far more appropriate to a functional game than .NET's or even F#'s mutable event systems), additional state values beyond the simulants shown above, and other types of dependencies. When you want something in your game to change, you operate on a World value to produce subsequent World values.

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 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 **BlazeDispatcher.createTitleScreen** 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

Proxies

As discussed before, simulants are not accessed or transformed directly, but rather through proxy types such as Entity, Group, Screen, et al. Proxies are created by using the proxy function respective to the desired type on an address that uniquely identifies the desired simulant. We'll elaborate more on addresses next.

Addresses

You may be wondering how the engine locates specific entities as created in NuEdit and loaded from the saved ***.nugroup** file. All entities, and other simulants, are located by constructing an **address** that uniquely identifies where it exists in an internal map in the engine. Each entity has an address of the form **ScreenName/GroupName/EntityName**, where **ScreenName** is the name that is given to its containing screen, **GroupName** is the name given to its 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 the **EntityName** portion of its address! The same structure applies to screen and groups addresses, albeit with fewer names. Game addresses are actually empty since there is only ever one game per world, thus no unique identifying information is needed.

Notice that addresses have a single type parameter that is used to make their intended usage more explicit. Addresses are used to both identify simulants as well as specify the events that take place upon them. You can tell the difference between simulant and event addresses by their type arguments, and even among different simulant and event types! Addresses used to locate simulants are typed according to the type of simulant they locate, and addresses that are used to specify events are typed according to the type of data their event carries.

For example, the **MouseMoveEventAddress** has a generic type of **MouseMoveData**, and an **EntityAddress** has a generic type of **EntityState**. Additionally, there are several operators and conversion functions used to combine addresses and manipulate their type appropriately in the **Address.fs** and **WorldAddress.fs** files of the **Nu** project. With these functions, you can combine simulant addresses with the common event address value found in **WorldConstants.fs** to specify event addresses as needed. This may all initially seem a little complicated, but please trust that this extra specificity it will save you from innumerable runtime errors.

The Purely-Functional Event System

Because the event system that F# provides out of the box is inherently mutating / impure, I was forced to invent a custom, 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.monitor**. You will likely be using this more often than the other two functions as it more compactly provides the desired behavior.

TODO: more detail!

Xtensions

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

Understanding the Xtension Type

Perhaps the most efficient way to exemplify the usage of an Xtension is by discussing its unit 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 this.GetDensity world : single = (this.GetXtension world)?Density
    member this.SetDensity (value : single) world = this.UpdateXtension (fun xtension -> xtension?Density <- value) world
```

- which when used in practice look like this –

```
    let world = entity.SetDensity 1.0f world
```

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

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
             define? ViewType Relative
             define? PublishChanges true
             define? Persistent true]

        /// Register an entity when adding it to a group.
        abstract Register : Entity -> World -> World
        default dispatcher.Register _ world = world

        /// Unregister an entity when removing it from a group.
        abstract Unregister : Entity -> World -> World
        default dispatcher.Unregister _ world = world

        /// Propagate an entity's physics properties from the physics subsystem.
        abstract PropagatePhysics : Entity -> World -> World
        default dispatcher.PropagatePhysics _ world = world

        /// Get the render descriptors needed to render an entity.
        abstract GetRenderDescriptors : Entity -> World -> RenderDescriptor list
        default dispatcher.GetRenderDescriptors _ _ = []
```

```
        /// Get the quick size of an entity (the appropriate user-define size for an entity).
        abstract GetQuickSize : Entity -> World -> Vector2
        default dispatcher.GetQuickSize _ _ = Vector2.One

        /// Get the priority with which an entity is picked in the editor.
        abstract GetPickingPriority : Entity -> single -> World -> single
        default dispatcher.GetPickingPriority _ depth _ = 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 this.GetSpriteImage world : AssetTag = (this.GetXtension world)?SpriteImage
        member this.SetSpriteImage (value : AssetTag) world = this.UpdateXtension (fun xtension -> xtension?SpriteImage <- value) world

    type SpriteFacet () =
        inherit Facet ()

        static member FieldDefinitions =
            [define? SpriteImage { PackageName = DefaultPackageName; AssetName = "Image3" }]

        override facet.GetRenderDescriptors entity world =
            if Camera.inView3 (entity.GetViewType world) (entity.GetPosition world) (entity.GetSize world) world.State.Camera then
                [LayerableDescriptor
                    { Depth = entity.GetDepth world
                      LayeredDescriptor =
                        SpriteDescriptor
                            { Position = entity.GetPosition world
                              Size = entity.GetSize world
                              Rotation = entity.GetRotation world
                              ViewType = entity.GetViewType world
                              OptInset = None
                              Image = entity.GetSpriteImage world
                              Color = Vector4.One }}]
            else []

        override facet.GetQuickSize entity world =
            match Metadata.tryGetTextureSizeAsVector2 (entity.GetSpriteImage world) 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

        // TODO: consider implementing Awake, AngularVelocity, and LinearVelocity fields to make
        // physics resets less destructive.

        member this.GetMinorId world : Guid = (this.GetXtension world)?MinorId
```

```fsharp
        member this.SetMinorId (value : Guid) world = this.UpdateXtension (fun xtension -> xtension?MinorId <- value) world
        member this.GetBodyType world : BodyType = (this.GetXtension world)?BodyType
        member this.SetBodyType (value : BodyType) world = this.UpdateXtension (fun xtension -> xtension?BodyType <- value) world
        member this.GetDensity world : single = (this.GetXtension world)?Density
        member this.SetDensity (value : single) world = this.UpdateXtension (fun xtension -> xtension?Density <- value) world
        member this.GetFriction world : single = (this.GetXtension world)?Friction
        member this.SetFriction (value : single) world = this.UpdateXtension (fun xtension -> xtension?Friction <- value) world
        member this.GetRestitution world : single = (this.GetXtension world)?Restitution
        member this.SetRestitution (value : single) world = this.UpdateXtension (fun xtension -> xtension?Restitution <- value) world
        member this.GetFixedRotation world : bool = (this.GetXtension world)?FixedRotation
        member this.SetFixedRotation (value : bool) world = this.UpdateXtension (fun xtension -> xtension?FixedRotation <- value) world
        member this.GetLinearDamping world : single = (this.GetXtension world)?LinearDamping
        member this.SetLinearDamping (value : single) world = this.UpdateXtension (fun xtension -> xtension?LinearDamping <- value) world
        member this.GetAngularDamping world : single = (this.GetXtension world)?AngularDamping
        member this.SetAngularDamping (value : single) world = this.UpdateXtension (fun xtension -> xtension?AngularDamping <- value) world
        member this.GetGravityScale world : single = (this.GetXtension world)?GravityScale
        member this.SetGravityScale (value : single) world = this.UpdateXtension (fun xtension -> xtension?GravityScale <- value) world
        member this.GetCollisionCategories world : string = (this.GetXtension world)?CollisionCategories
        member this.SetCollisionCategories (value : string) world = this.UpdateXtension (fun xtension -> xtension?CollisionCategories <- value) world
        member this.GetCollisionMask world : string = (this.GetXtension world)?CollisionMask
        member this.SetCollisionMask (value : string) world = this.UpdateXtension (fun xtension -> xtension?CollisionMask <- value) world
        member this.GetCollisionExpr world : string = (this.GetXtension world)?CollisionExpr
        member this.SetCollisionExpr (value : string) world = this.UpdateXtension (fun xtension -> xtension?CollisionExpr <- value) world
        member this.GetIsBullet world : bool = (this.GetXtension world)?IsBullet
        member this.SetIsBullet (value : bool) world = this.UpdateXtension (fun xtension -> xtension?IsBullet <- value) world
        member this.GetIsSensor world : bool = (this.GetXtension world)?IsSensor
        member this.SetIsSensor (value : bool) world = this.UpdateXtension (fun xtension -> xtension?IsSensor <- value) world
        member this.GetPhysicsId world = { SourceId = this.GetId world; BodyId = this.GetMinorId world }

    type RigidBodyFacet () =
        inherit Facet ()

        static let getBodyShape (entity : Entity) world =
            Physics.evalCollisionExpr (entity.GetSize world) (entity.GetCollisionExpr world)

        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? CollisionExpr "Box"
             define? IsBullet false
             define? IsSensor false]

        override facet.RegisterPhysics entity world =
            let bodyProperties =
                { BodyId = (entity.GetPhysicsId world).BodyId
                  Position = entity.GetPosition world + entity.GetSize world * 0.5f
                  Rotation = entity.GetRotation world
                  Shape = getBodyShape entity world
                  BodyType = entity.GetBodyType world
                  Density = entity.GetDensity world
                  Friction = entity.GetFriction world
                  Restitution = entity.GetRestitution world
                  FixedRotation = entity.GetFixedRotation world
                  LinearDamping = entity.GetLinearDamping world
                  AngularDamping = entity.GetAngularDamping world
                  GravityScale = entity.GetGravityScale world
                  CollisionCategories = Physics.toCollisionCategories <| entity.GetCollisionCategories world
                  CollisionMask = Physics.toCollisionCategories <| entity.GetCollisionMask world
                  IsBullet = entity.GetIsBullet world
                  IsSensor = entity.GetIsSensor world }
            World.createBody entity.EntityAddress (entity.GetId world) bodyProperties world

        override facet.UnregisterPhysics entity world =
            World.destroyBody (entity.GetPhysicsId world) world

        override facet.PropagatePhysics entity world =
            let world = facet.UnregisterPhysics entity world
            facet.RegisterPhysics 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 –

```fsharp
[<AutoOpen>]
module RigidSpriteDispatcherModule =

    type RigidSpriteDispatcher () =
        inherit EntityDispatcher ()

        static member IntrinsicFacetNames =
            [typeof<RigidBodyFacet>.Name
             typeof<SpriteFacet>.Name]
```
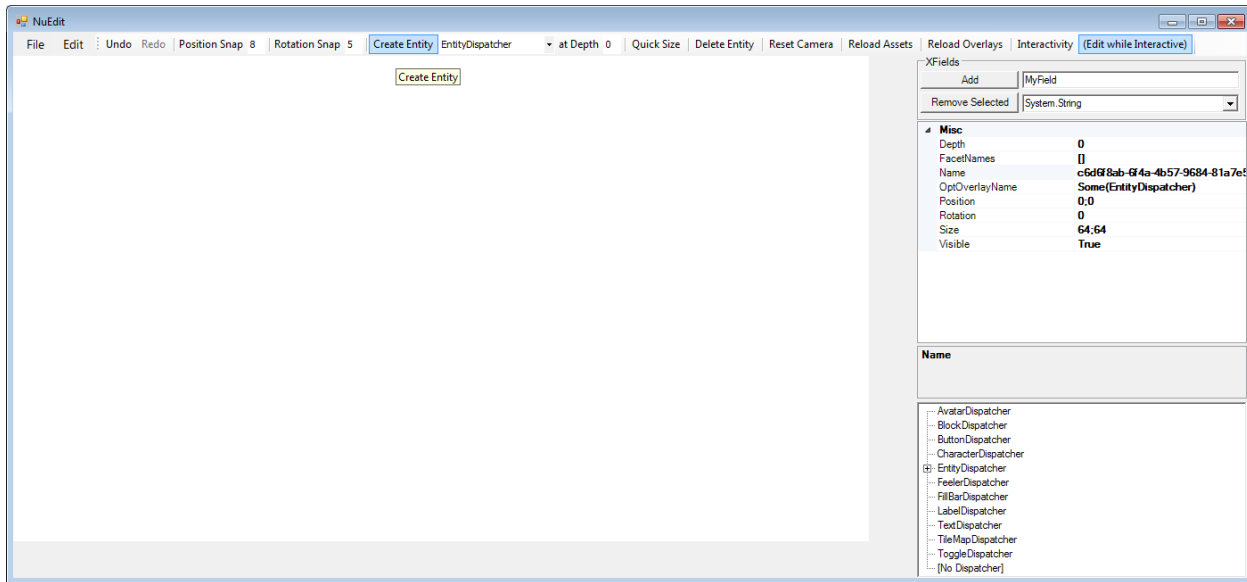
Additionally, facets can be dynamically added to a removed from an entity in NuEdit simply by changing the FacetNames property. Let's take a look.

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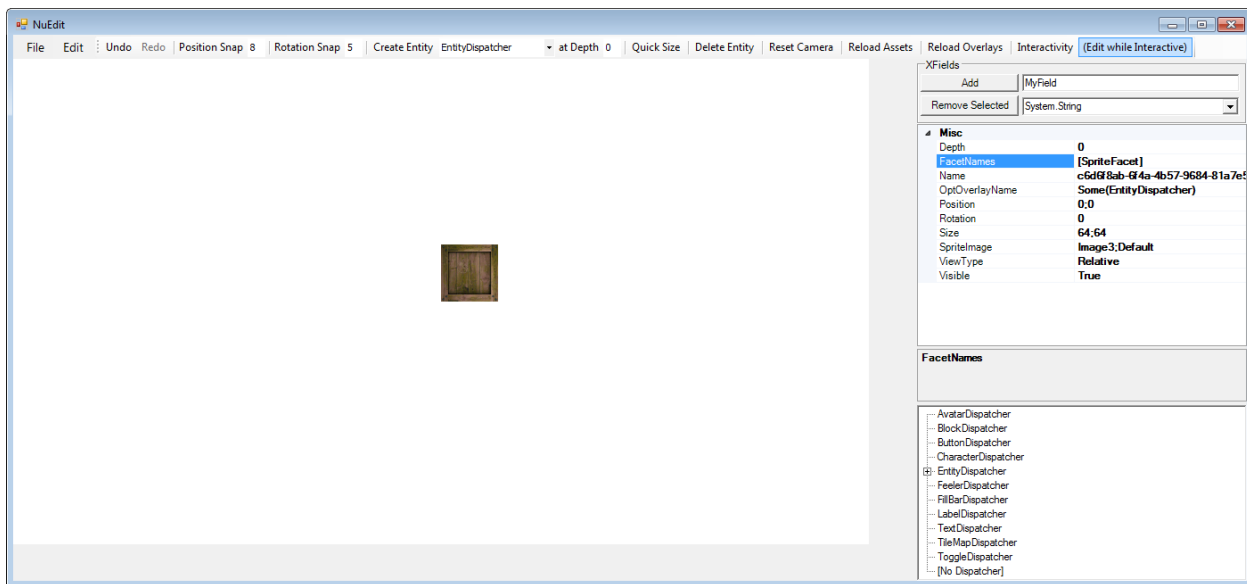
Not only does it now render, the additional fields needed to specify how rendering is performed are provided in the property grid (to the right). Try adding physics to the entity by changing **FacetNames** to **[SpriteFacet | RigidBodyFacet]**.

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.
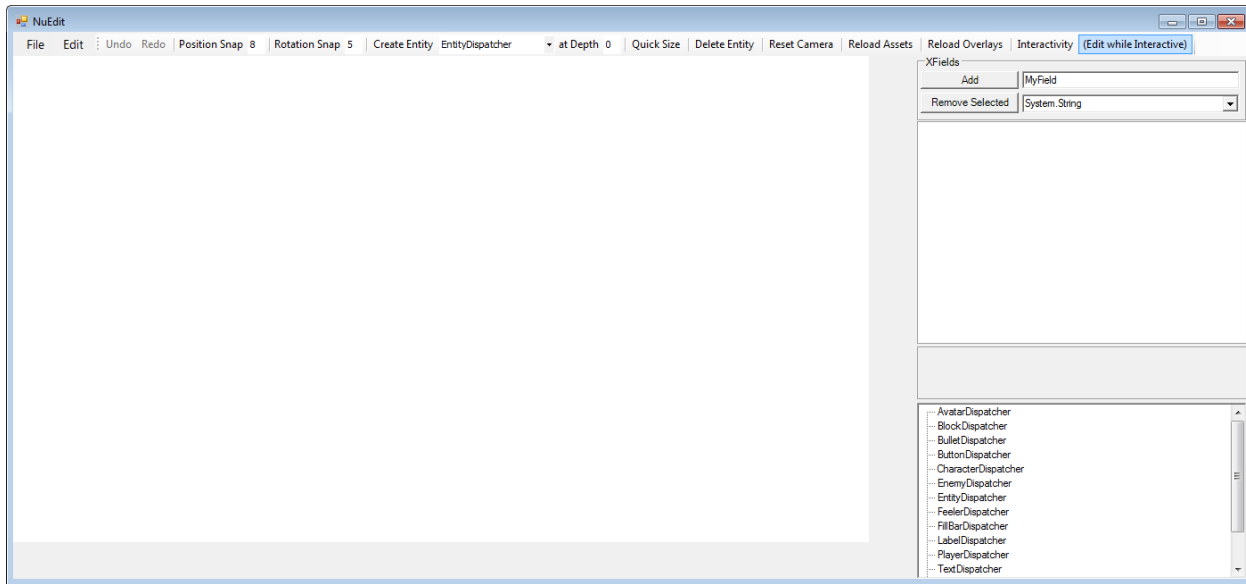
Bullets and the BulletDispatcher

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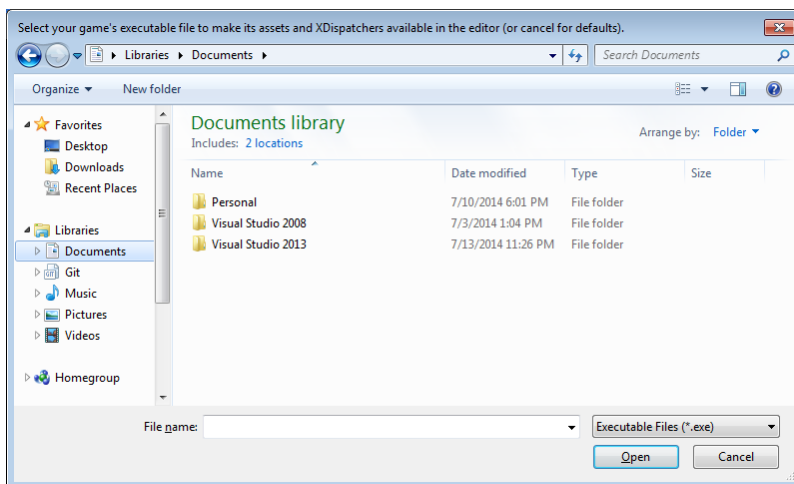
Since we need the **BulletDispatcher** from the BlazeVector.exe file, navigate the Open File dialog to the **./BlazeVector/BlazeVector/BlazeVector/bin/Debug** folder and select the **BlazeVector.exe** file. The editor will now open up as normal –

- 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 –
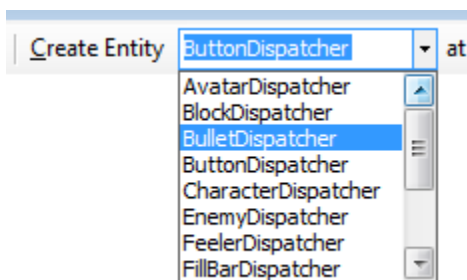
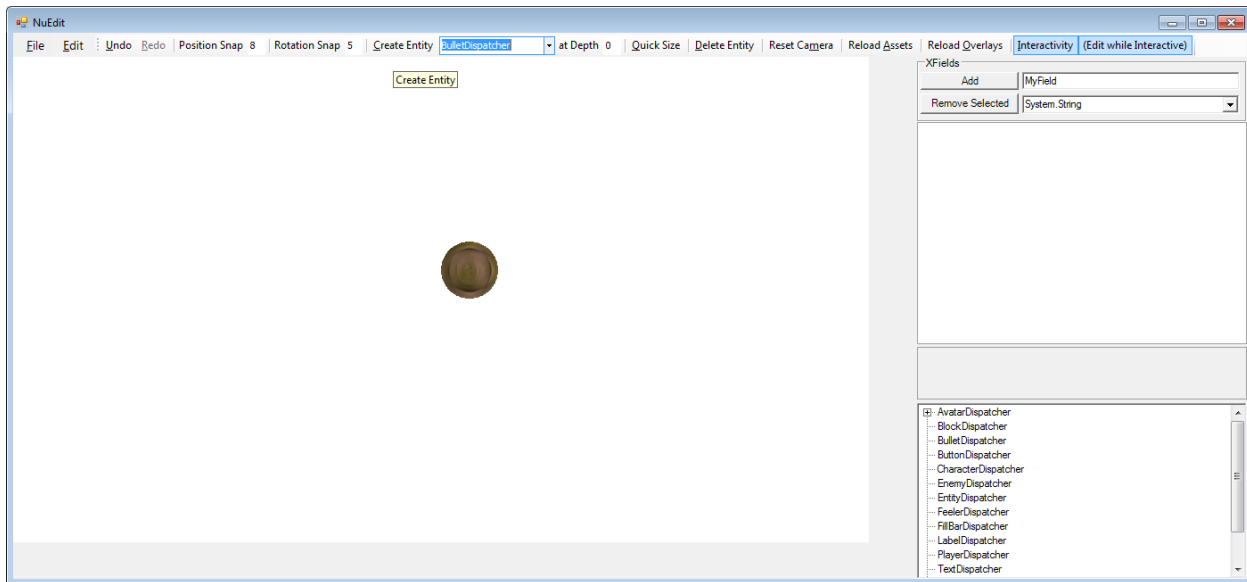Let's select the **BulletDispatcher** option and click **Create Entity** to create a bullet entity like so –
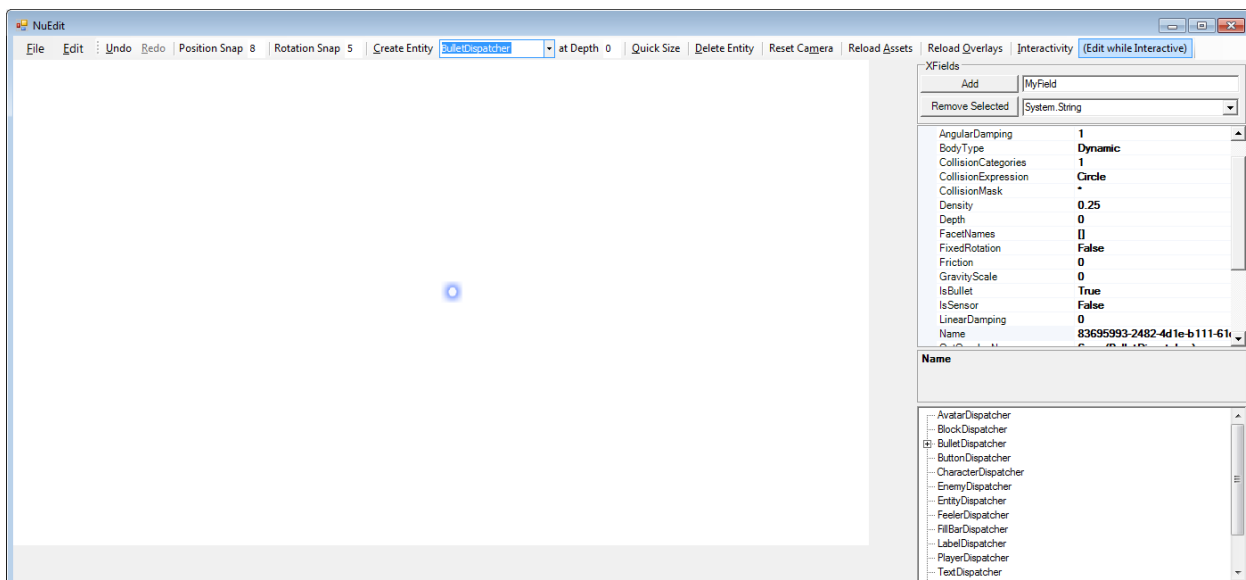
The bullet doesn't really have much behavior, but that's because the **Interactivity** button (top-left) is toggled off. Let's try toggling it on.

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 **TopViewCharacterDispatcher**, and then click **Create Entity**. You should end up with this –

Since the character 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 character 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 BulletModule =

    type Entity with

        member this.GetAge world : int64 = (this.GetXtension world)?Age
        member this.SetAge (value : int64) world = this.UpdateXtension (fun xtension -> xtension?Age <- value) world

    type BulletDispatcher () =
        inherit EntityDispatcher ()

        static let [<Literal>] BulletLifetime = 27L

        static let handleUpdate event world =
            let bullet = event.Subscriber : Entity
            let world = bullet.SetAge (bullet.GetAge world + World.getTickRate world) world
```

```
            let world =
                if bullet.GetAge world > BulletLifetime
                then World.destroyEntity bullet world
                else world
            (Cascade, world)

        static let handleCollision event world =
            let bullet = event.Subscriber : Entity
            if World.isTicking world then
                let world = World.destroyEntity bullet world
                (Cascade, world)
            else (Cascade, 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? CollisionExpr "Circle"
             define? SpriteImage PlayerBulletImage
             define? Age 0L]

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

        override dispatcher.Register bullet world =
            world |>
                World.monitor handleTick TickEventAddress bullet |>
                World.monitor handleCollision (CollisionEventAddress ->>- bullet.EntityAddress) bullet
```

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 this.GetAge world : int64 = (this.GetXtension world)?Age
    member this.SetAge (value : int64) world = this.UpdateXtension (fun xtension -> xtension?Age <- value) world
```

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. An int64 is used because that is the type Nu prefers for values that represent time.

```
type 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? CollisionExpr "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 such as in the editor.

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 bullet world =
    world |>
        World.monitor handleUpdate UpdateEventAddress bullet |>
        World.monitor handleCollision (CollisionEventAddress ->>- bullet.EntityAddress) bullet
```

The first line specifies that we're overriding the **Register** method of EntityDispatcher. Typically, there is no need to call the **base.Register** methods for types that inherit directly from dispatchers or facets. The second and third lines are used to monitor and react to certain events for the duration of the entity's lifetime. The dispatcher's **handleTick** function will be called whenever the tick event is published (see the previous section **The Purely-Functional Event System**), and the dispatcher's **handleCollision** will be called whenever a collision takes place on the bullet.

The handleUpdate

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

```
static let handleUpdate event world =
    let bullet = event.Subscriber : Entity
    let world = bullet.SetAge (bullet.GetAge world + World.getTickRate world) world
    let world =
        if bullet.GetAge world > BulletLifetime
        then World.destroyEntity bullet world
        else world
    (Cascade, world)
```

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

The first line simply aliases the bullet proxy from the **event.Subscriber** field for convenience. You could use the event fields directly, but I find such aliasing to help me keep things straight in my head. YMMV!

The next line increments the bullet's **Age** field according to the world's **TickRate**. The TickRate is how fast the simulation is progressing, where 0 is stopped, 1 is normal, 2 is twice speed, etc.

Lines 3 - 6 check if the bullet is older than **BulletLifetime** ticks (**27**), and then destroys the bullet only if so.

The handleCollision

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

```
static let handleCollision event world =
    let bullet = event.Subscriber : Entity
    if World.isTicking world then
        let world = World.destroyEntity bullet world
        (Cascade, world)
    else (Cascade, world)
```

Even simpler than the previous handler, it merely destroys the bullet when a collision with it takes place while the world is ticking (EG – the world's TickRate is not zero).

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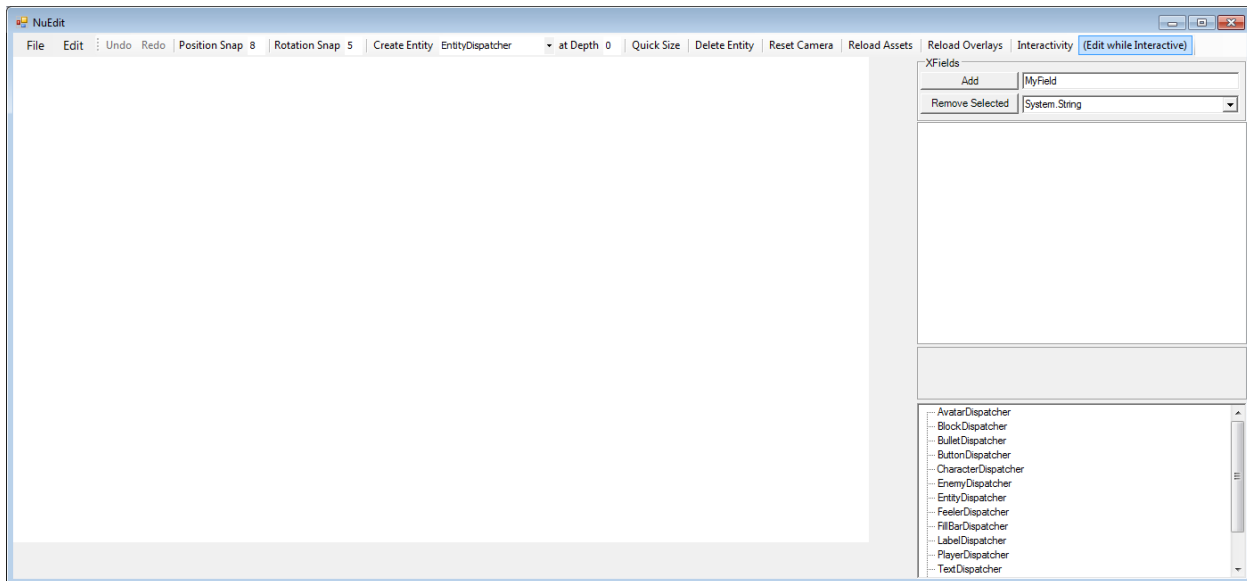
Before creating enemies, let's create a tile map in which for them to live by selecting the **TileMapDispatcher**, clicking **Create Entity**, and then dragging the bottom-left corner of the tile map to the bottom left corner of the screen -
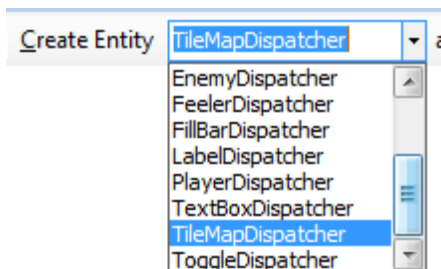

We will end up with something like this –
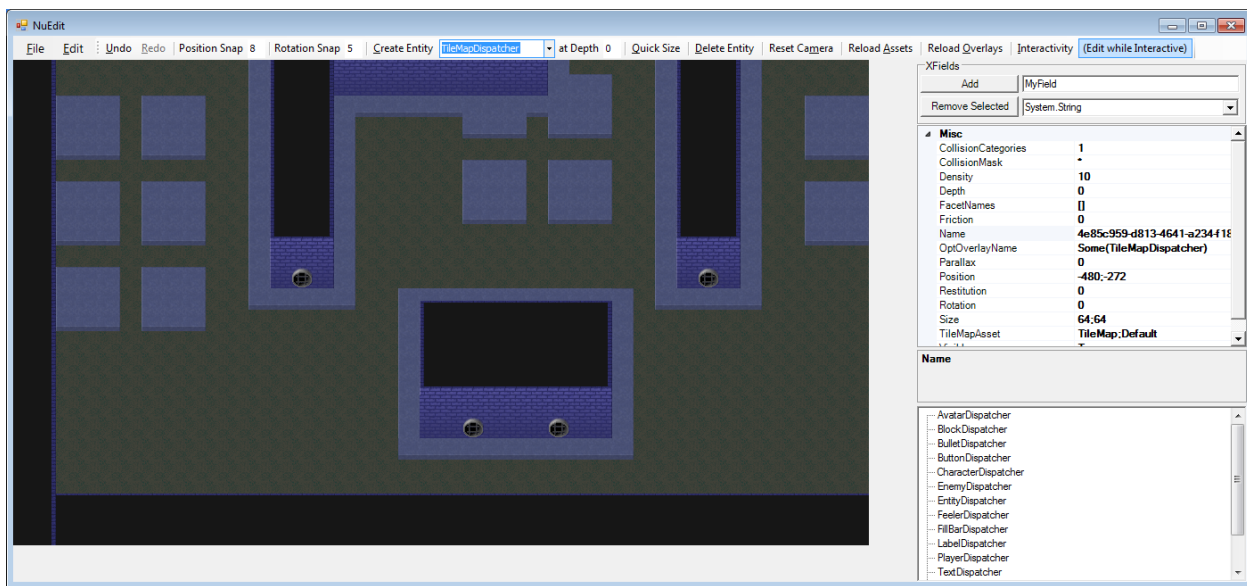
Since we want to create additional entities on top of the bottom layer of the tile map (but below the second layer), we set **at Depth** value in the editor's tool bar to **1**. Next, we create a few enemies by selecting the **Enemy Dispatcher** and clicking **Create Entity** a few times (or alternatively, 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 –
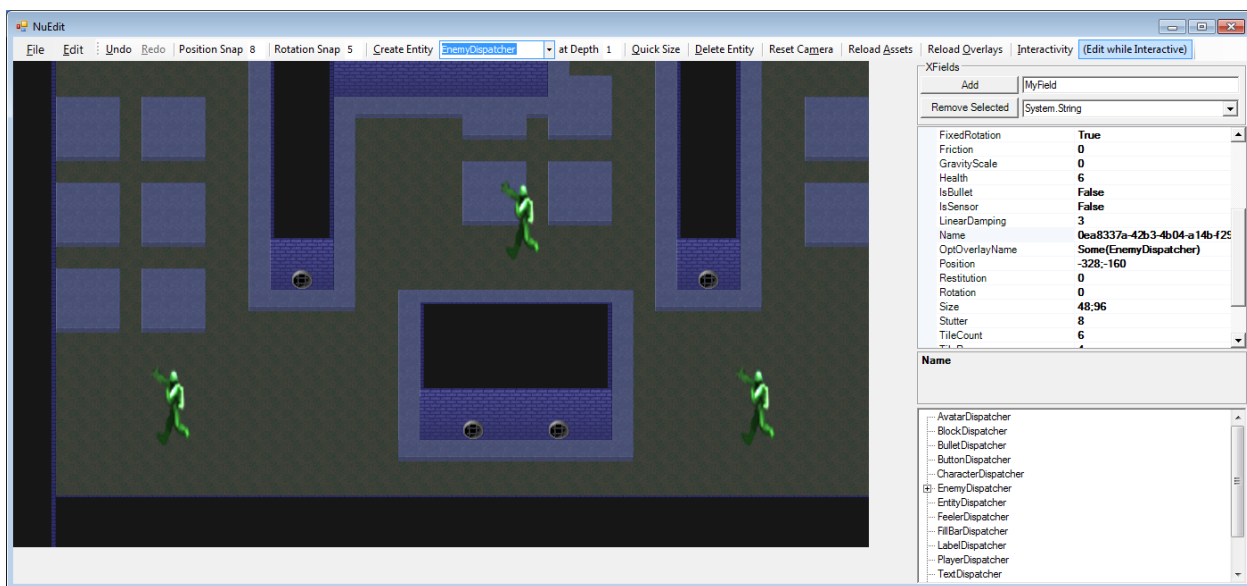
Now, for fun, let's toggle the **Interactivity** button to see what these enemies would do during gameplay.
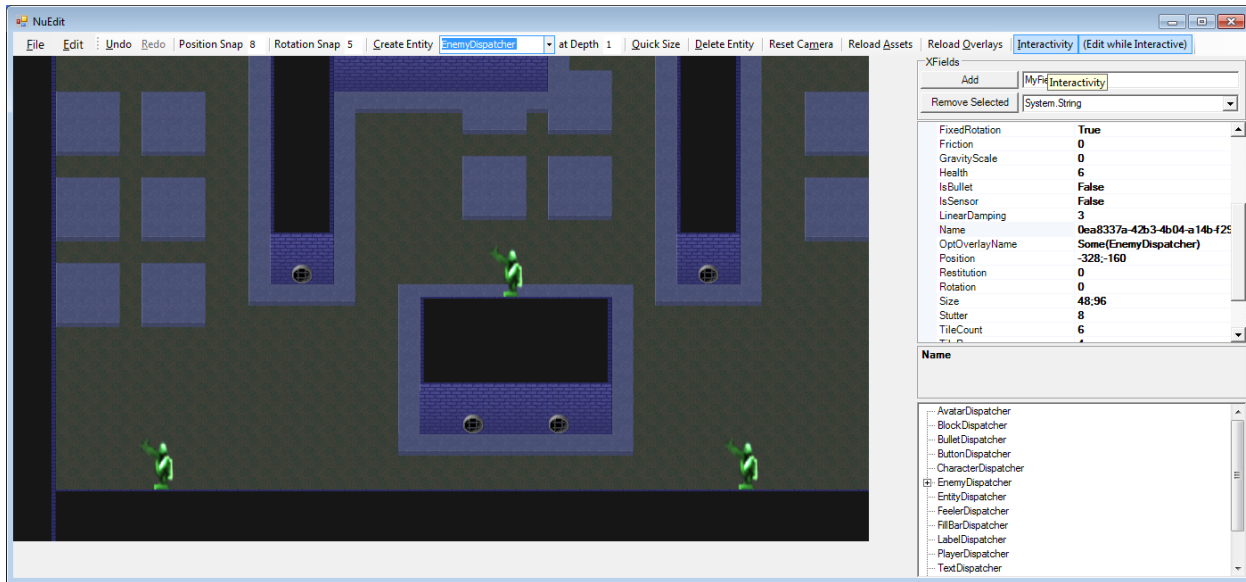
They just drop to the ground and walk to the left!

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)\" "$(TargetDir)\" "$(ProjectDir)Refinement" False

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="Font" file="Assets/Default/FreeMonoBold.032.ttf" associations="[Render]"/>
    <Assets directory="Assets/Default" extension="png" associations="[Render]"/>
    <Assets directory="Assets/Default" extension="wav" associations="[Audio]"/>
    <Assets directory="Assets/Default" extension="ogg" associations="[Audio]"/>
```

```
        <Assets directory="Assets/Default" extension="tmx"/>
    </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 **[Default | Image]** (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 render 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.hintRenderPackageUse "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 render 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.hintRenderPackageDisuse** 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 –
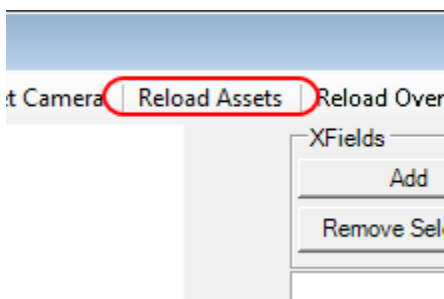
Serialization and Overlays

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.

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

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

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

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

  <SampleDispatcherOverlay includes="[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 Gui, 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) –

```xml
<MyButtonOverlay includes="[ButtonDispatcher]">
  <Enabled>False</Enabled>
  <OptClickSound>[Some [GuiPackage | MyClickSound]]</OptClickSound>
</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 **includes** attribute in the Overlay.xml file (like is done with **includes="[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.

TODO: Cover use of the OverlayRouter!

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.hintRenderPackageUse** 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 **Render** subsystem, an **Audio** subsystem, and a **Physics** subsystem. Additional subsystems such as **Artificial Intelligence**, **Particles**, and et al can be added by overriding the **MakeSubsystems** method in your **NuPlugin** subtype.