Nu Game Engine

The practical and efficient functional game engine!

Copyright © Bryan Edds 2013, 2019

What's It All About?

The Nu Game Engine is a **Mature**, **Functional**, **2d Game Engine** written in **F#**.

Let me explain each of those terms –

Mature

Nu is mature, however, it is still missing a few frills. For example, there is no AI system for scripting intelligent simulants yet, nor a high-performance particle system. 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. There is also a special effects system called, appropriately enough, **EffectSystem**. On top of all that, there is a built-in game editor called **Gaia**! So while there are some missing features, you can see they might be worth waiting for, or even building for yourself!

Functional

Nu is built mostly 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. Users can opt-out to impure semantics for added efficiency (see https://github.com/bryanedds/Nu/blob/b10d25339a523e43c4fa95fc55fde7e0963668da/Projects/Metrics/Program.fs#L10-L16 on how to configure an entity for optimization). Additionally, there are a lot imperative operations going on behind the scenes for speed! For example, the Farseer Physics system is written in an imperative style in C#, and many parts of the engine are optimized with imperative caches 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 unless you explicitly opt-out for speed.

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 runs on both the .NET Framework and Windows, and with a bit of project tinkering, on Mono and Linux. I'm hoping to soon port it to .NET Core. 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 Haskell, you get intuitive and a well-tooled debugging experience. 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/Nu. 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 install the VC 2012 redistributable **vcredist_x64** (link is here - https://www.microsoft.com/en-in/download/details.aspx?id=30679)

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 a **Projects** folder containing the sample game **BlazeVector** (which we'll be studying in this document), my WIP role-playing game **InfinityRpg**, and others.

To open the Nu solution, first make sure to have **Visual Studio 2017** installed (the free edition is fine). Then open the **Nu.sln** file in the root folder. 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 directly via bryanedds@gmail.com.

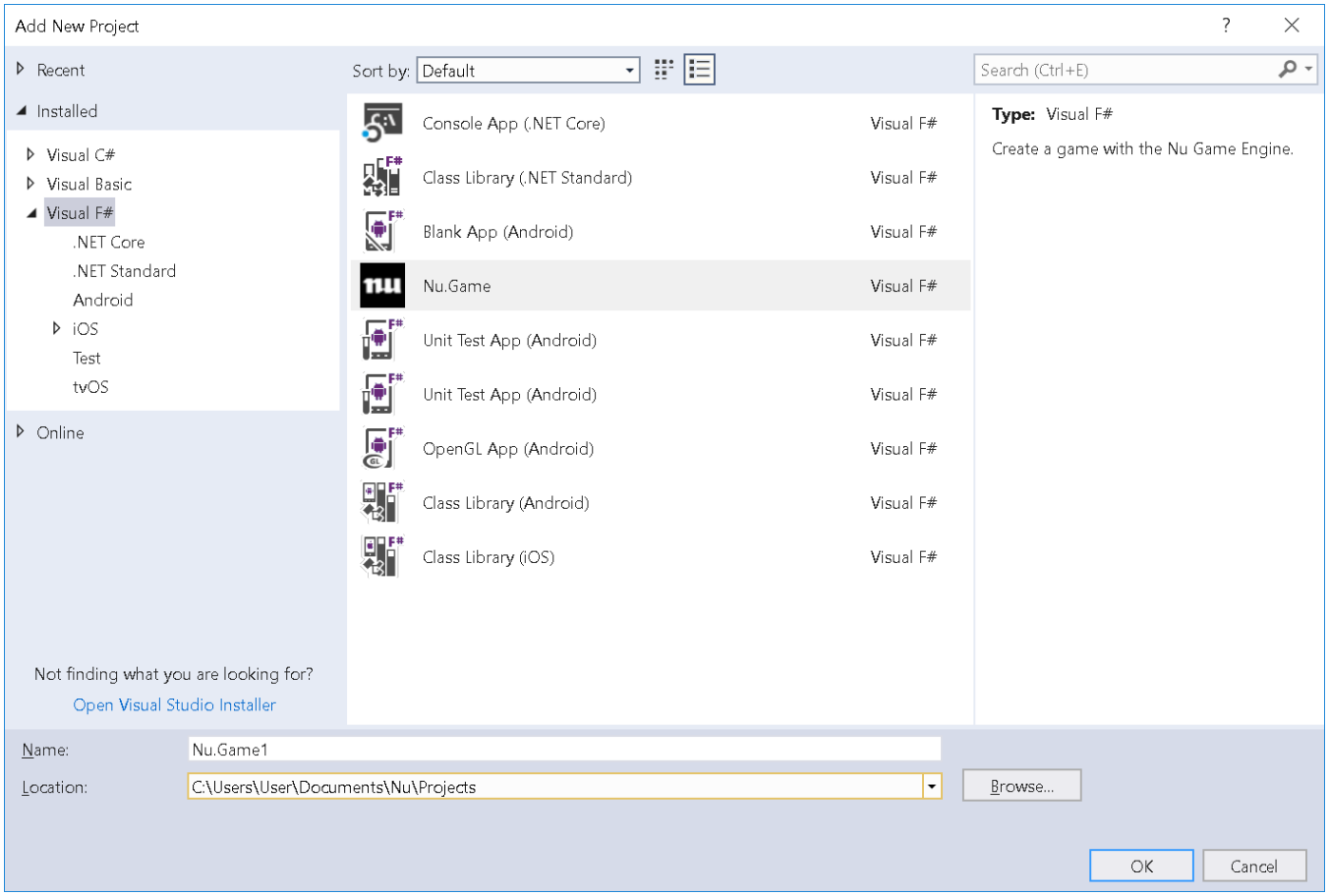
Once the solution builds successfully, ensure that the **BlazeVector** project is set as the **StartUp** project, and then run the game by pressing the **|> Start** button in Visual Studio.

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.Template.Export** 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 –



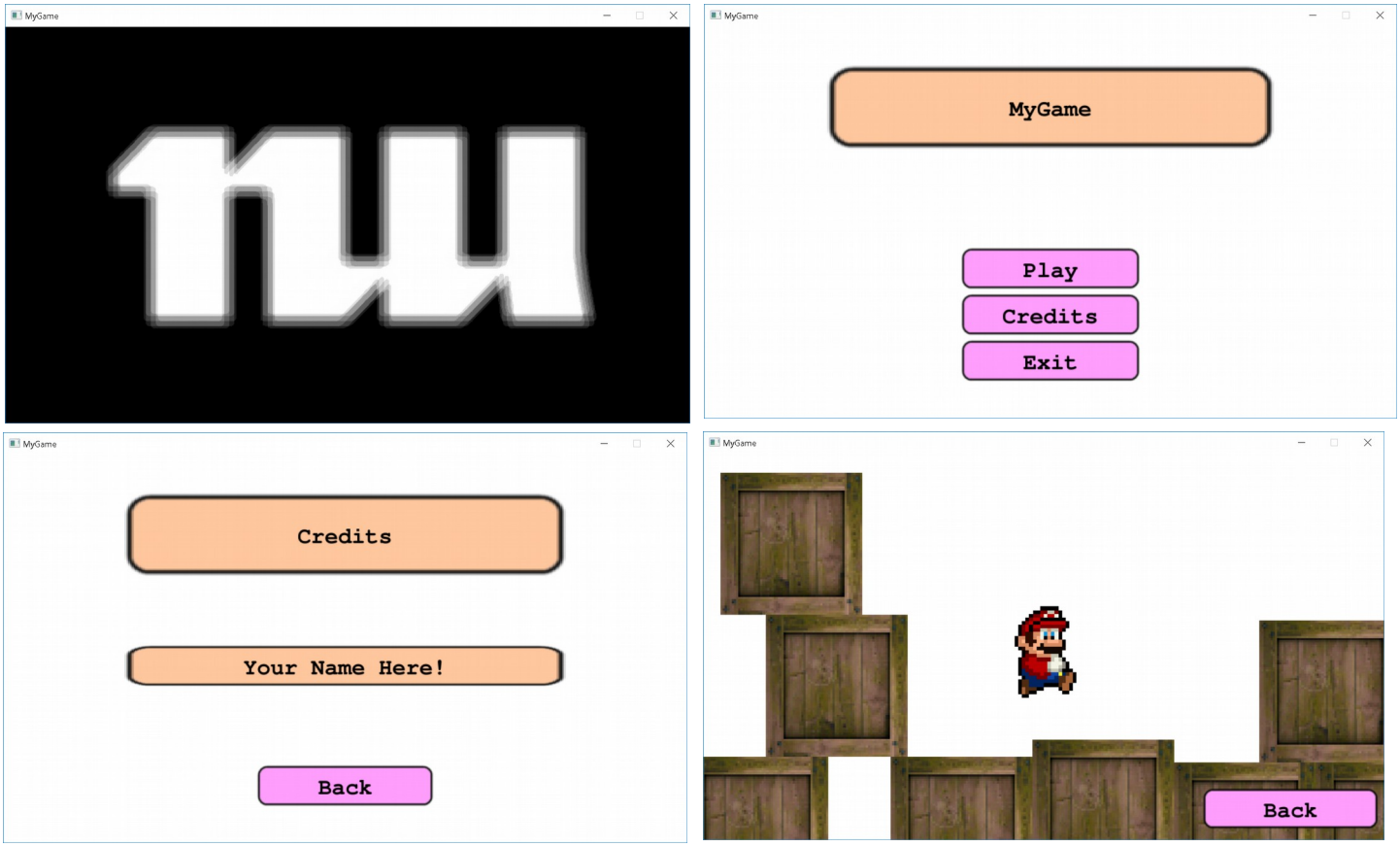
- and enter the name of your game in the **Name** text box.

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 ☺

Finally, set the Location field to the provided **./Nu/Projects** folder as above. Note that if this is done incorrectly, the new project will not be able to find the Nu, Nu.Pipe, and SDL2-CS dependencies needed to build it!

With everything configured as above, click the **OK** button to create the project. Now you can build and run 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 get the basic template game that includes a splash screen, a title screen, a credits screen, and a little mario-like gameplay screen



Basic Nu Start-up Code

Here's the start-up code presented with comments in **Program.fs** -

```
namespace MyGame
open System
open Nu

// this is a plugin for the Nu game engine by which user-defined dispatchers, facets, and other
// sorts of types can be obtained by both your application and Gaia.
type MyPlugin () =
    inherit NuPlugin ()

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

    // make our game-specific game dispatchers...
    override this.MakeGameDispatchers () =
        [MyGameDispatcher () :> GameDispatcher]

    // specify the above game dispatcher to use at run-time
    override this.GetStandAloneGameDispatcherName () =
        typeof<MyGameDispatcher>.Name

    // specify the sceen dispatcher to optionally use in the editor
    override this.GetGameplayScreenDispatcherName () =
        typeof<MyGameplayDispatcher>.Name

    // specify the empty game dispatcher to use in the editor
    override this.GetEditorGameDispatcherName () =
```

```
        typeof<GameDispatcher>.Name

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

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

        // initialize Nu
        Nu.init false

        // this specifies the window configuration used to display the game.
        let sdlWindowConfig = { SdlWindowConfig.defaultConfig with WindowTitle = "MyGame" }

        // this specifies the configuration of the game engine's use of SDL.
        let sdlConfig = { SdlConfig.defaultConfig with ViewConfig = NewWindow sdlWindowConfig }

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

            // an instance of the above plugin
            let plugin = MyPlugin ()

            // here is an attempt to make the world with the various initial states, the engine
            // plugin, and SDL dependencies.
            World.tryMake true 1L () plugin sdlDeps

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

All this code in the first half of the document does is set up the plug-in that makes your custom types available for instantiation in the game and in Nu's game editor, Gaia. The second half merely initializes Nu and instantiates the game engine with the above plug-in.

Let's take a some code that is more interesting in **MyGame.fs** -

```
namespace MyGame
open Prime
open Nu

// this is our Elm-style command type. To learn about the Elm-style, read this article here -
// https://medium.com/@bryanedds/a-game-engine-that-allows-you-to-program-in-the-elm-style-31d806fbe27f
type MyGameCommand =
    | ShowTitle
    | ShowCredits
    | ShowGameplay
    | ExitGame

// this is the game dispatcher that is customized for our game. In here, we create screens and wire
// them up with events and transitions.
type MyGameDispatcher () =
    inherit GameDispatcher<unit, unit, MyGameCommand> ()

    // here we define the Bindings used to connect events to their desired commands
    override dispatcher.Bindings (_, _, _) =
        [Simulants.TitleCredits.ClickEvent =>! ShowCredits
         Simulants.TitlePlay.ClickEvent =>! ShowGameplay
         Simulants.TitleExit.ClickEvent =>! ExitGame
         Simulants.CreditsBack.ClickEvent =>! ShowTitle
         Simulants.Back.ClickEvent =>! ShowTitle]

    // here we handle the above commands
    override dispatcher.Command (command, _, _, world) =
        match command with
        | ShowTitle -> World.transitionScreen Simulants.Title world
```

```
        | ShowCredits -> World.transitionScreen Simulants.Credits world
        | ShowGameplay -> World.transitionScreen Simulants.Gameplay world
        | ExitGame -> World.exit world
```

```
    // here we describe the content of the game including all of its screens.
    override dispatcher.Content (_, _, _) =
        [Content.screen Simulants.Splash.Name (Splash (Default.DissolveData, Default.SplashData, Simulants.Title)) [] []
         Content.screenFromLayerFile Simulants.Title.Name (Dissolve Default.DissolveData) "Assets/Gui/Title.nulyr"
         Content.screenFromLayerFile Simulants.Credits.Name (Dissolve Default.DissolveData)
"Assets/Gui/Credits.nulyr"
         Content.screen<MyGameplayDispatcher> Simulants.Gameplay.Name (Dissolve Default.DissolveData) [] []]

    // here we hint to the renderer and audio system that the 'Gui' package should be loaded ahead of time
    override dispatcher.Register (game, world) =
        let world = World.hintRenderPackageUse "Gui" world
        let world = World.hintAudioPackageUse "Gui" world
        base.Register (game, world)
```

As mentioned in the comments, the best way to understand this code is to first read my article here -

https://medium.com/@bryanedds/a-game-engine-that-allows-you-to-program-in-the-elm-style-31d806fbe27f

It explains how you can program Nu simulants in the Elm-style, which is how the above code is programmed.

Once you understand how Nu is programmed in the Elm-style, let's take a look at how we actually define the gameplay portion of the game in **MyGameplay.fs** -

```
namespace MyGame
open Prime
open SDL2
open Nu
open Nu.Declarative

// this is our Elm-style command type
type GameplayCommand =
    | Jump
    | MoveLeft
    | MoveRight
    | EyeTrack
    | Nop

// this is the screen dispatcher that defines the screen where gameplay takes place.
type MyGameplayDispatcher () =
    inherit ScreenDispatcher<unit, unit, GameplayCommand> ()

    // here we define the Bindings used to connect events to their desired commands
    override this.Bindings (_, _, _) =
        [// here we issue a command for the game's camera eye to track the player at all times
         Simulants.Gameplay.UpdateEvent =>! EyeTrack

         // here we issue a command for the player to jump when the up arrow is pressed
         Simulants.Game.KeyboardKeyDownEvent =|>! fun evt ->
            if evt.Data.ScanCode = int SDL.SDL_Scancode.SDL_SCANCODE_UP && not evt.Data.Repeated
            then Jump
            else Nop

         // here we issue a command for the player to walk when the left or right arrows are pressed
         Simulants.Gameplay.UpdateEvent =|>! fun _ ->
            if KeyboardState.isKeyDown (int SDL.SDL_Scancode.SDL_SCANCODE_LEFT) then MoveLeft
            elif KeyboardState.isKeyDown (int SDL.SDL_Scancode.SDL_SCANCODE_RIGHT) then MoveRight
            else Nop]

    // here we handle the above commands
    override this.Command (command, _, _, world) =
        match command with
        | Jump ->
            // when the jump command is issued, we apply an upward physics force to the player so
            // long as the player's body is on the ground.
            let physicsId = Simulants.Player.GetPhysicsId world
            if World.isBodyOnGround physicsId world
            then World.applyBodyForce (v2 0.0f 2000000.0f) physicsId world
            else world
        | MoveLeft ->
            // here we apply a leftward force whose magnitude depends on whether the player's body
            // is on the ground
            let physicsId = Simulants.Player.GetPhysicsId world
            if World.isBodyOnGround physicsId world
            then World.applyBodyForce (v2 -30000.0f 0.0f) physicsId world
            else World.applyBodyForce (v2 -7500.0f 0.0f) physicsId world
        | MoveRight ->
            // same as above but with a rightward force
            let physicsId = Simulants.Player.GetPhysicsId world
            if World.isBodyOnGround physicsId world
            then World.applyBodyForce (v2 30000.0f 0.0f) physicsId world
            else World.applyBodyForce (v2 7500.0f 0.0f) physicsId world
        | EyeTrack ->
            // here we issue instructions for the camera to follow the player wherever he goes
            if World.getTickRate world <> 0L
```

```
            then Simulants.Game.SetEyeCenter (Simulants.Player.GetCenter world) world
            else world
        | Nop ->
            // nop is the empty command. Nothing is done here.
            world

    // here we describe the content of the game including the hud, the player, and the level
    override this.Content (_, _, _) =
        [// here we define the hud with just one button, the Back button
         Content.layer Simulants.Hud.Name []
            [Content.button Simulants.Back.Name
                [Entity.Text == "Back"
                 Entity.Position == v2 220.0f -260.0f
                 Entity.Depth == 10.0f]]

         // here we define the scene that includes the player. By wrapping the scene definition in
         // Content.layerIfScreenSelected, we make it exist only when the specified screen is the
         // current one. When that screen is no longer selected, the scene is deleted.
         Content.layerIfScreenSelected Simulants.Gameplay $ fun () ->
            Content.layer Simulants.Scene.Name []
                [Content.character Simulants.Player.Name
                    [Entity.Position == v2 0.0f 0.0f
                     Entity.Size == v2 144.0f 144.0f]]

         // here we define the level that is loaded from the specified asset.
         Content.layerIfScreenSelected Simulants.Gameplay $ fun () ->
            Content.layerFromFile Simulants.Level.Name "Assets/Gui/Level.nulyr"]
```

Finally, let's look at how the simulants themselves are structured in **MySimulants.fs** -

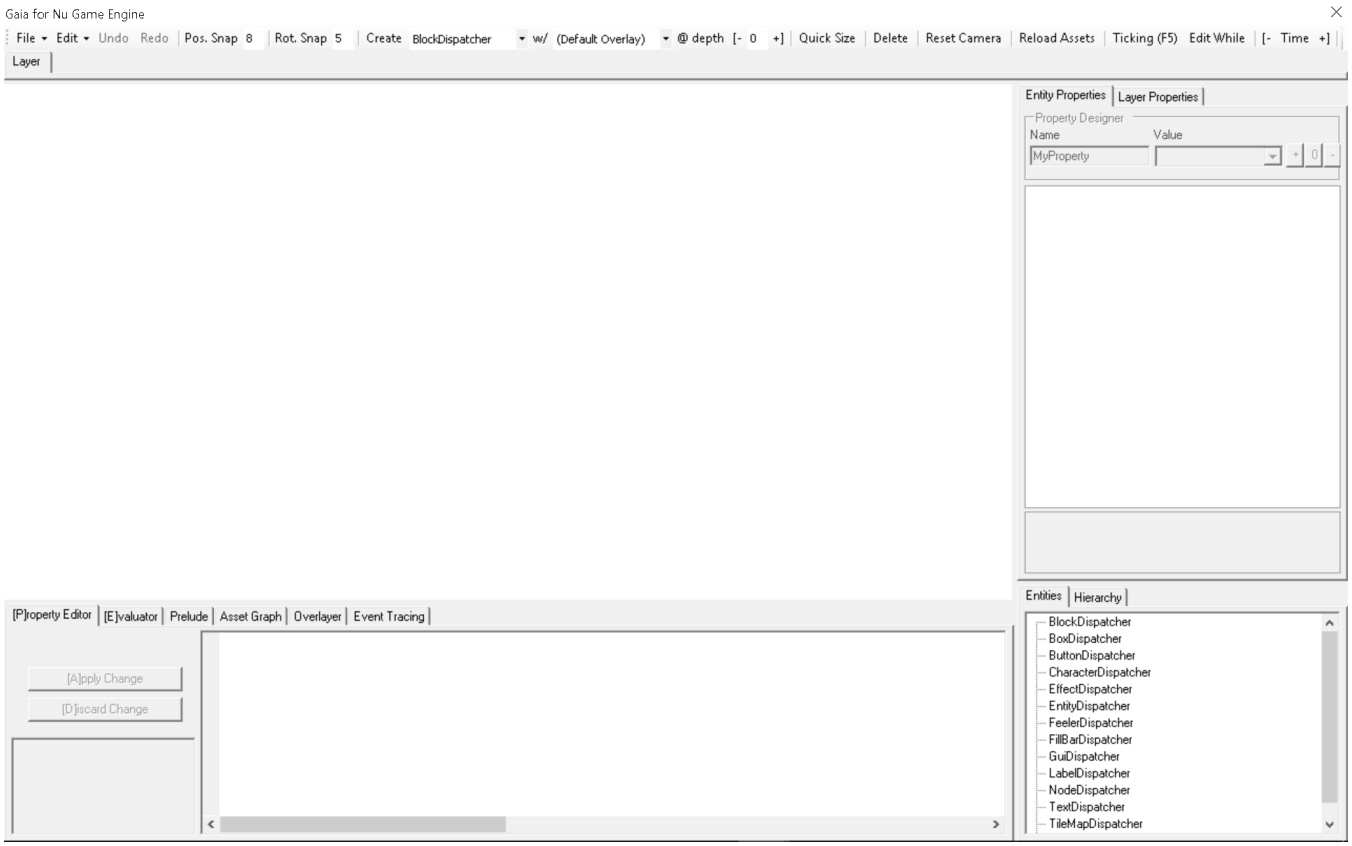
```
namespace MyGame
open Nu

[<RequireQualifiedAccess>]
module Simulants =

    // the handle for the game
    let Game = Default.Game

    // same as above, but for the splash screen
    let Splash = Screen "Splash"

    // same as above, but for the title screen and its children
    let Title = Screen "Title"
    let TitleGui = Title / "Gui"
    let TitlePlay = TitleGui / "Play"
    let TitleCredits = TitleGui / "Credits"
    let TitleExit = TitleGui / "Exit"

    // credits screen handles
    let Credits = Screen "Credits"
    let CreditsGui = Credits / "Gui"
    let CreditsBack = CreditsGui / "Back"

    // gameplay screen handles
    let Gameplay = Default.Screen
    let Hud = Gameplay / "Hud"
    let Back = Hud / "Back"
    let Scene = Gameplay / "Scene"
    let Player = Scene / "Player"
    let Level = Gameplay / "Level"
```

And that's all there is to it!

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, **Gaia**.

What is Gaia?

Gaia is Nu's game editing tool. Here is a screenshot of an empty editing session –

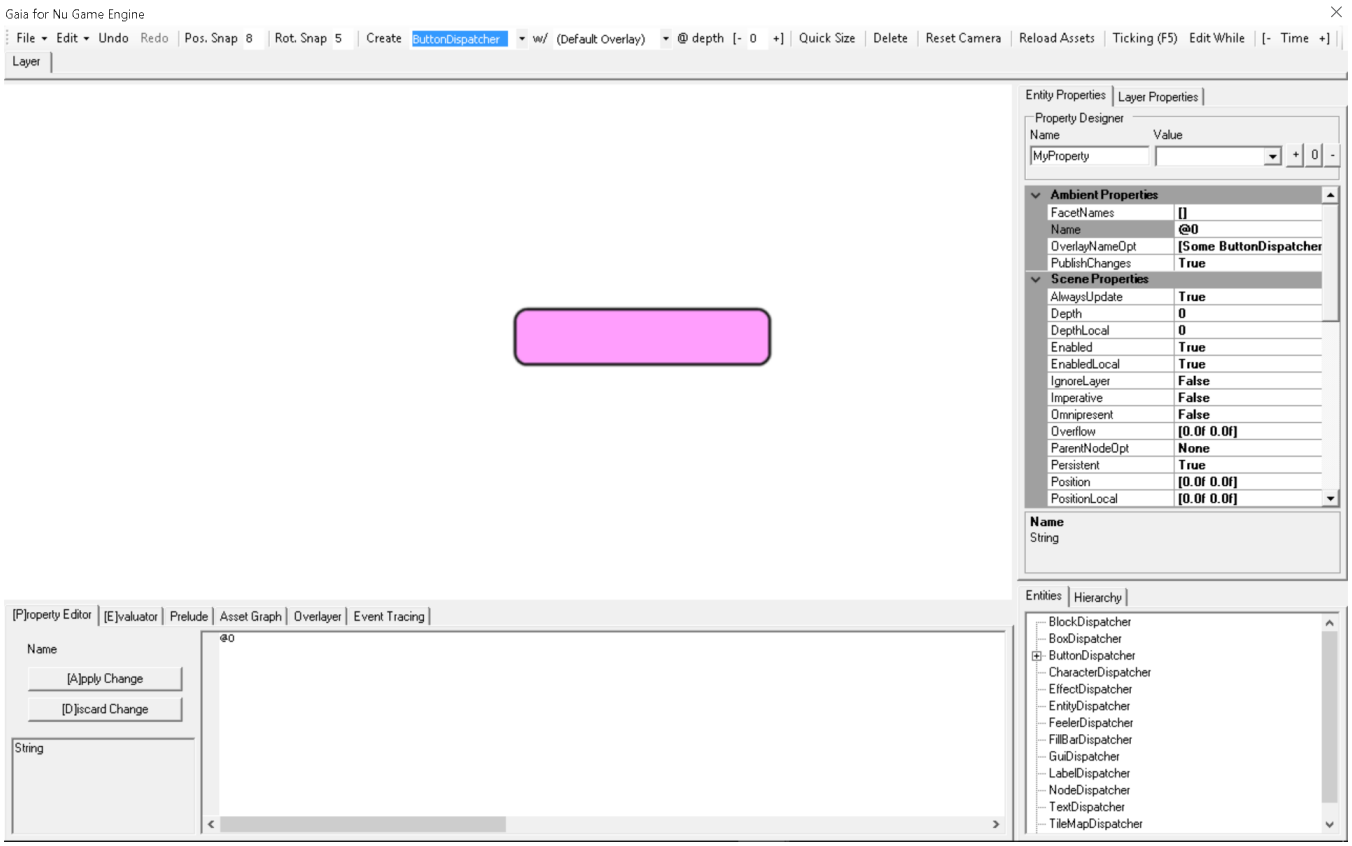


Run Gaia by setting the **Gaia project** as the **StartUp Project** in Visual Studio, and then running.

Upon starting, tou'll notice the **Editor Start Configuration** screen. If you select your game's .NET executable, the custom types that you exposed in your **Plugin** type will be available for use in the editor. If you cancel this dialog, you get only what comes with Nu out of the box. Additionally, you can have the editor open your gameplay screen (assuming you have it assigned to **Default.Screen** as in the above template game).

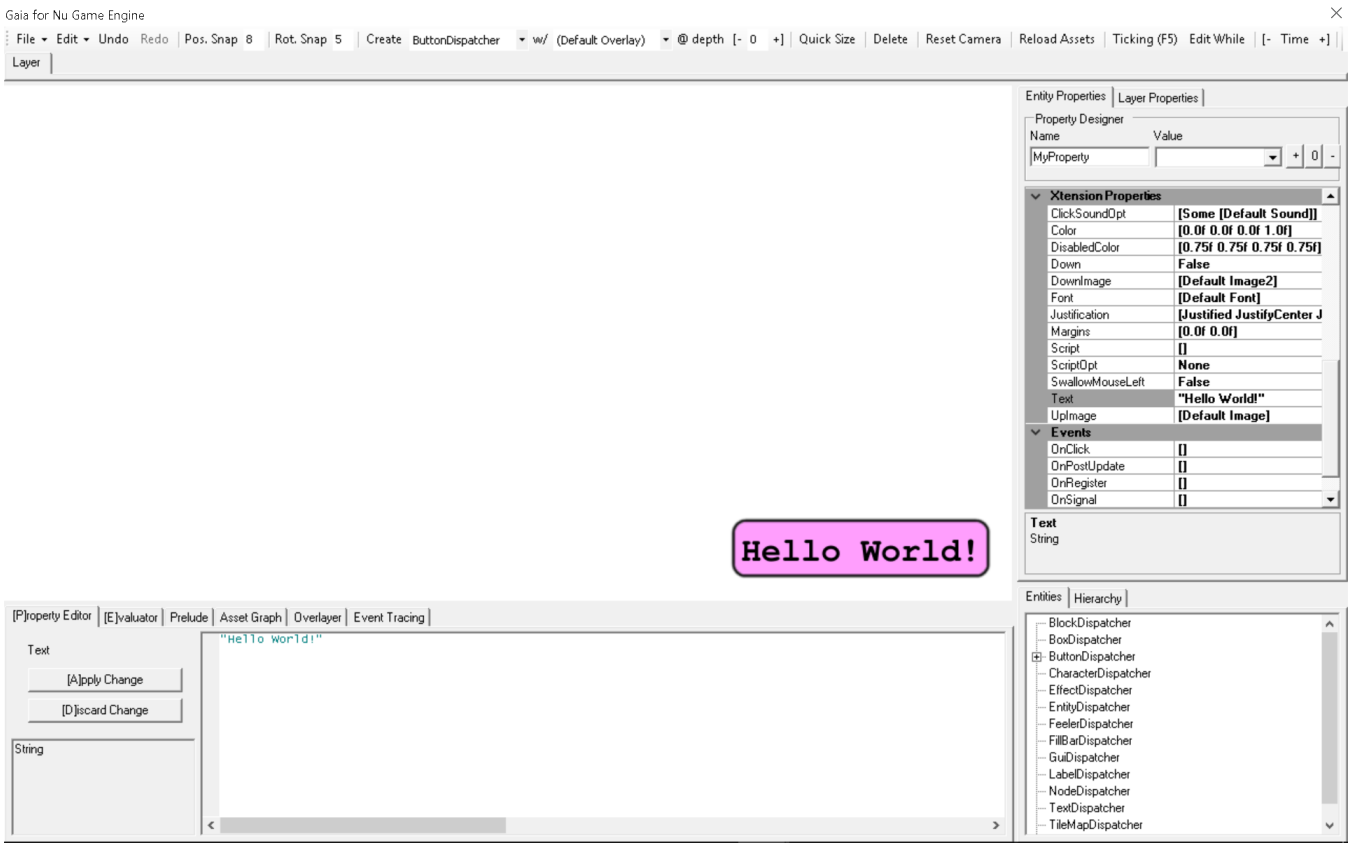
Here we will just cancel the dialog and play with the dispatchers / facets that come out-of-the-box.

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



We have a button! Now notice that the property grid on the right has been populated with its properties. These properties 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** property, deleting the contents, and then entering the text **MyButton**. Naming entities give you the ability to access them at run-time via that name once you have loaded the containing document in your game.

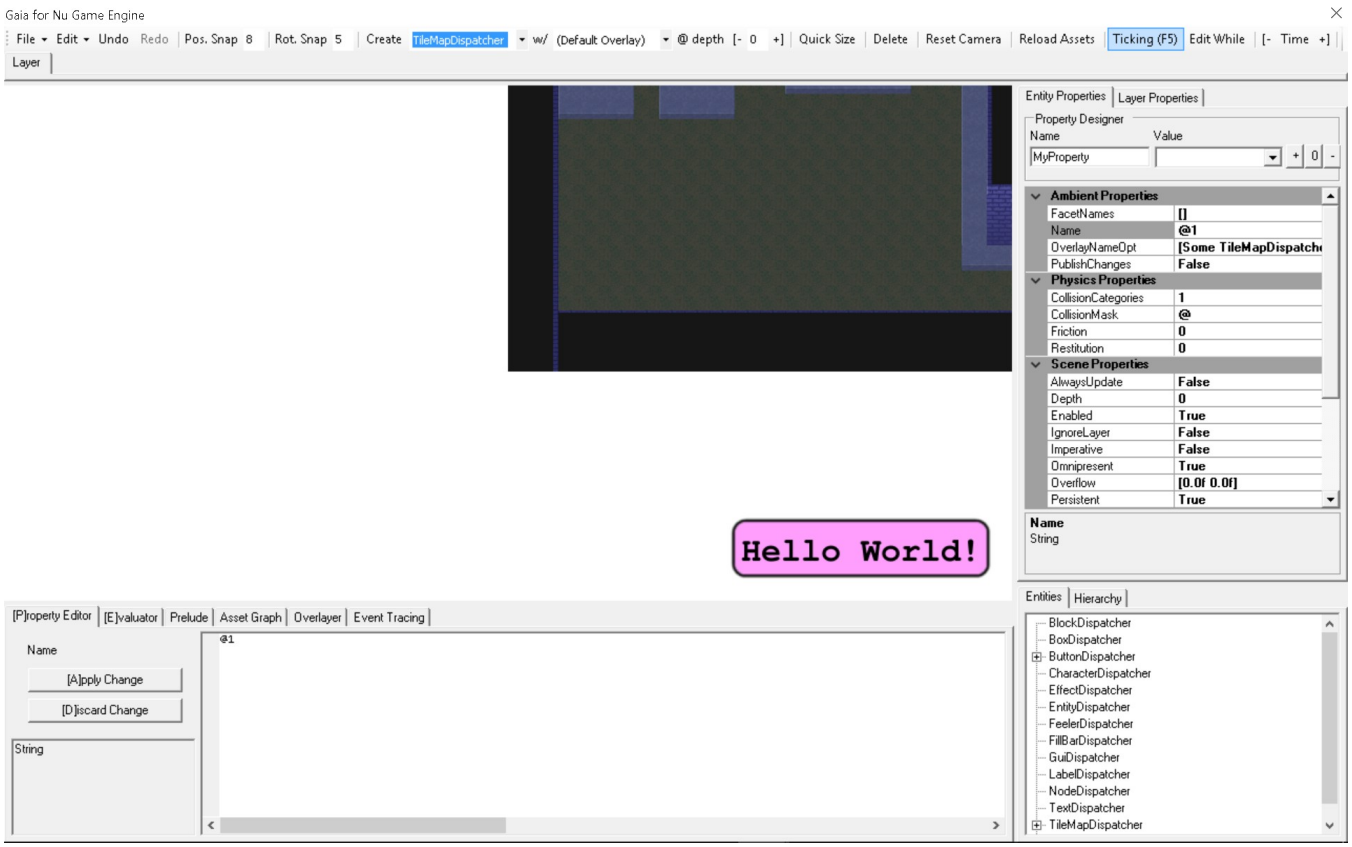
Notice also that you can click and drag on the button to move it about the screen. You can also right-click and entity for additional operations via a context menu. Let's change the **Text** property of the button to "Hello World!" and more the button to the bottom right of the screen -



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

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

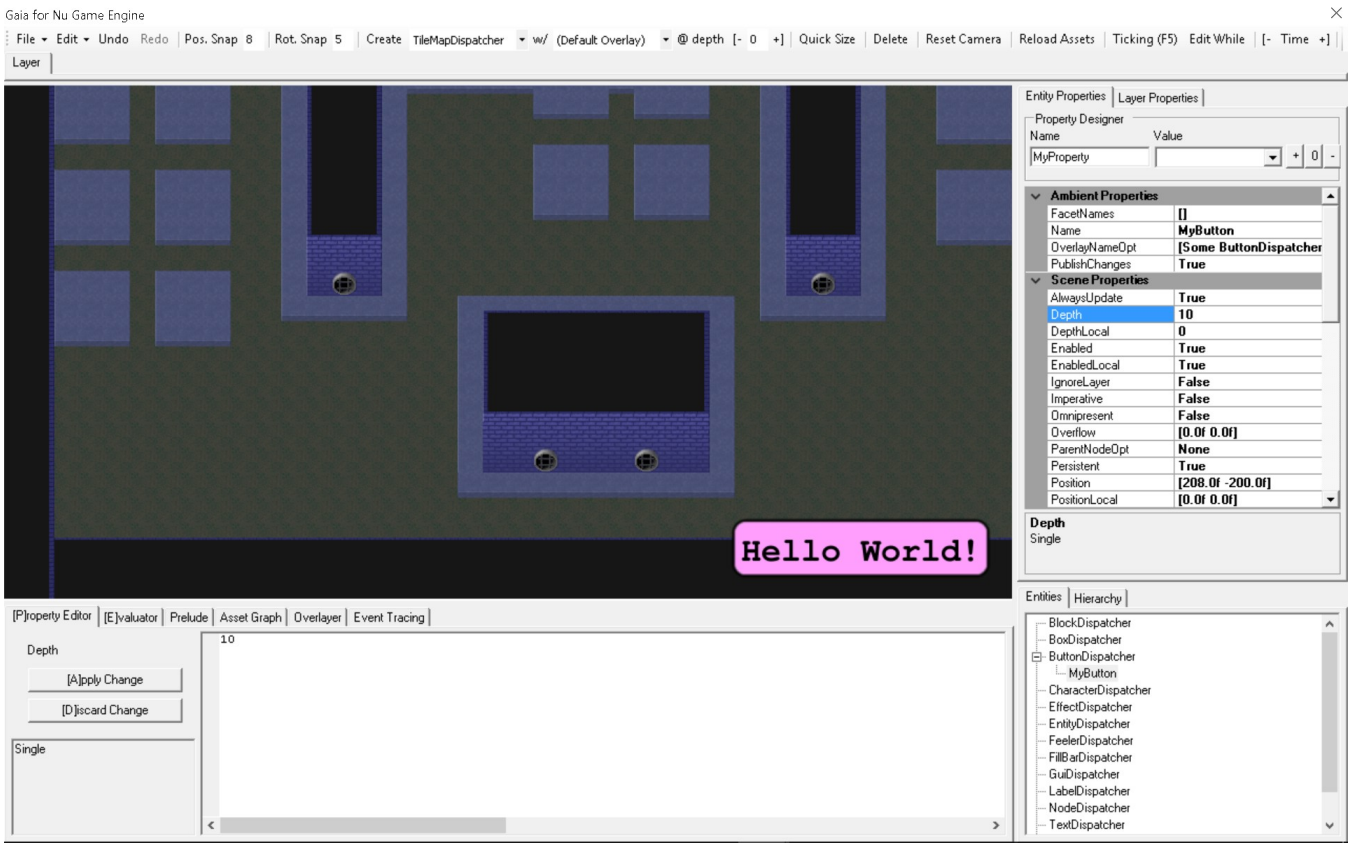
Now let's make a default tile map to play around with. BUT FIRST, we need to change the depth of our button entity so that it doesn't get covered by the new tile map. Change the value in the button's **Depth** property to **10**.



In the drop down box to the right of the **Create Entity** button, select (or type) **TileMapDispatcher**, and then press the **Create Entity** button. 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.

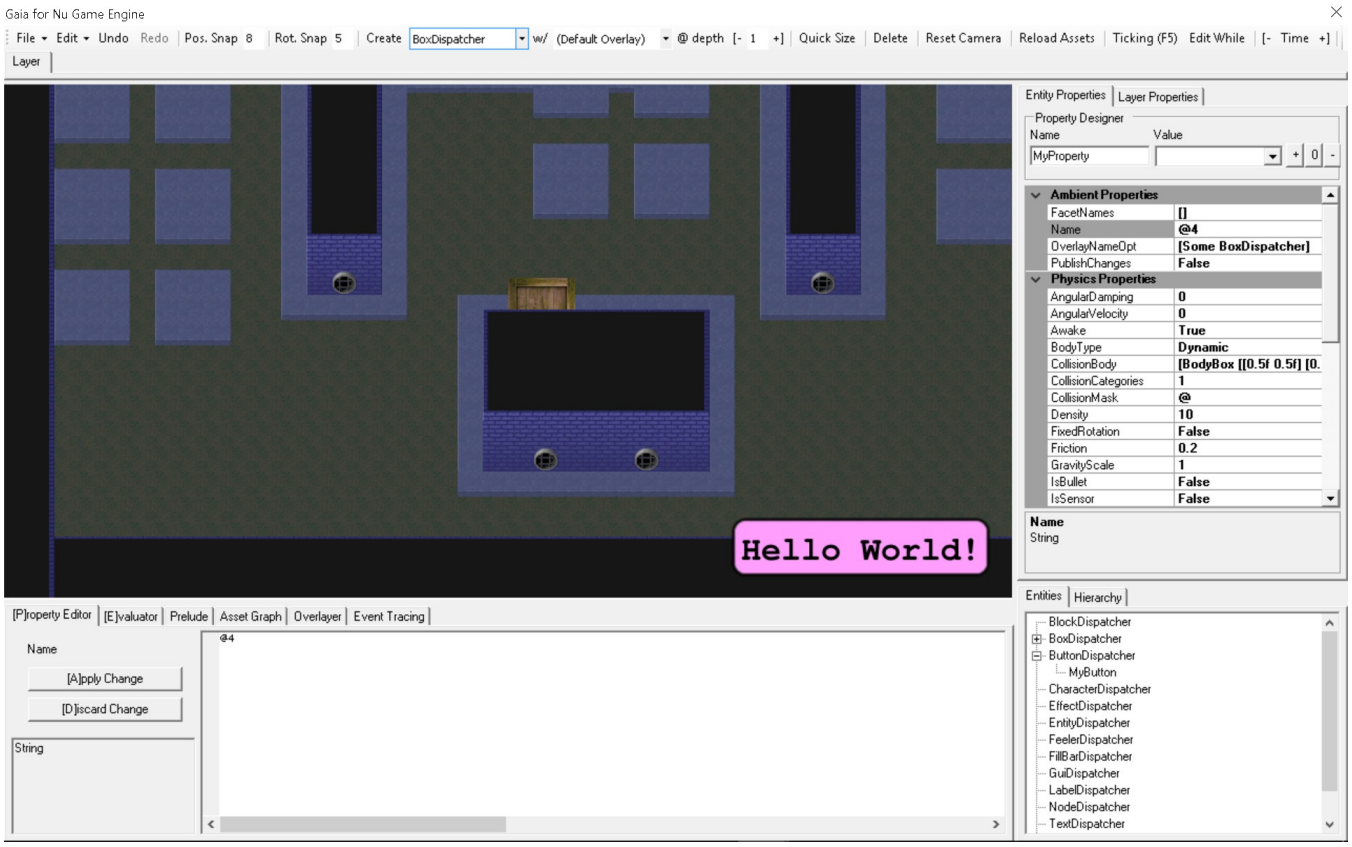
Tile maps, by the way, are created with the free tile map editor **Tiled** found at http://www.mapeditor.org/. All credit to the great chap who made and maintains it!



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 **Ticking** 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.



The **BoxDispatcher** is affected by gravity, so try moving the box upward and the letting it fall. Why does it not fall? Well, the physics system is not enabled unless the game is ticking. But according to the **Ticking** toggle button at the top left, ticking is not toggled on. So let us toggle it on, and watch the box fall according to gravity!

Turn off **Ticking** when you're satisfied.

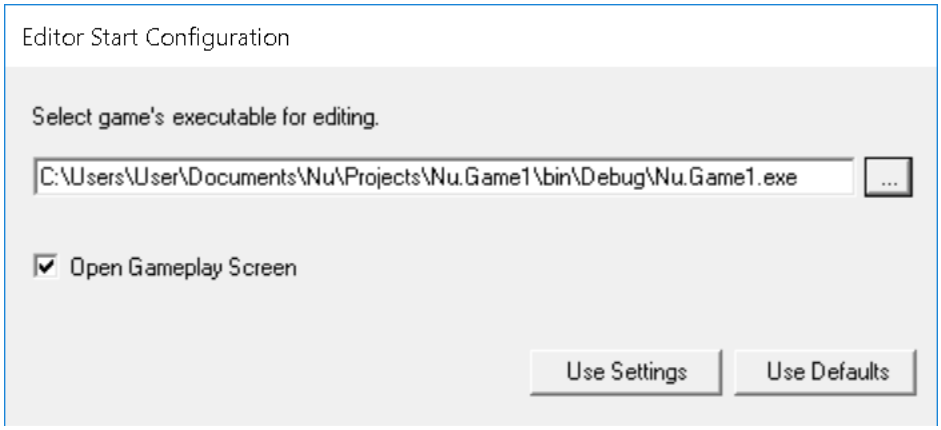
Another way to create boxes is by right-clicking at the desired location and then, in the context menu that pops up, clicking **Create**.

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

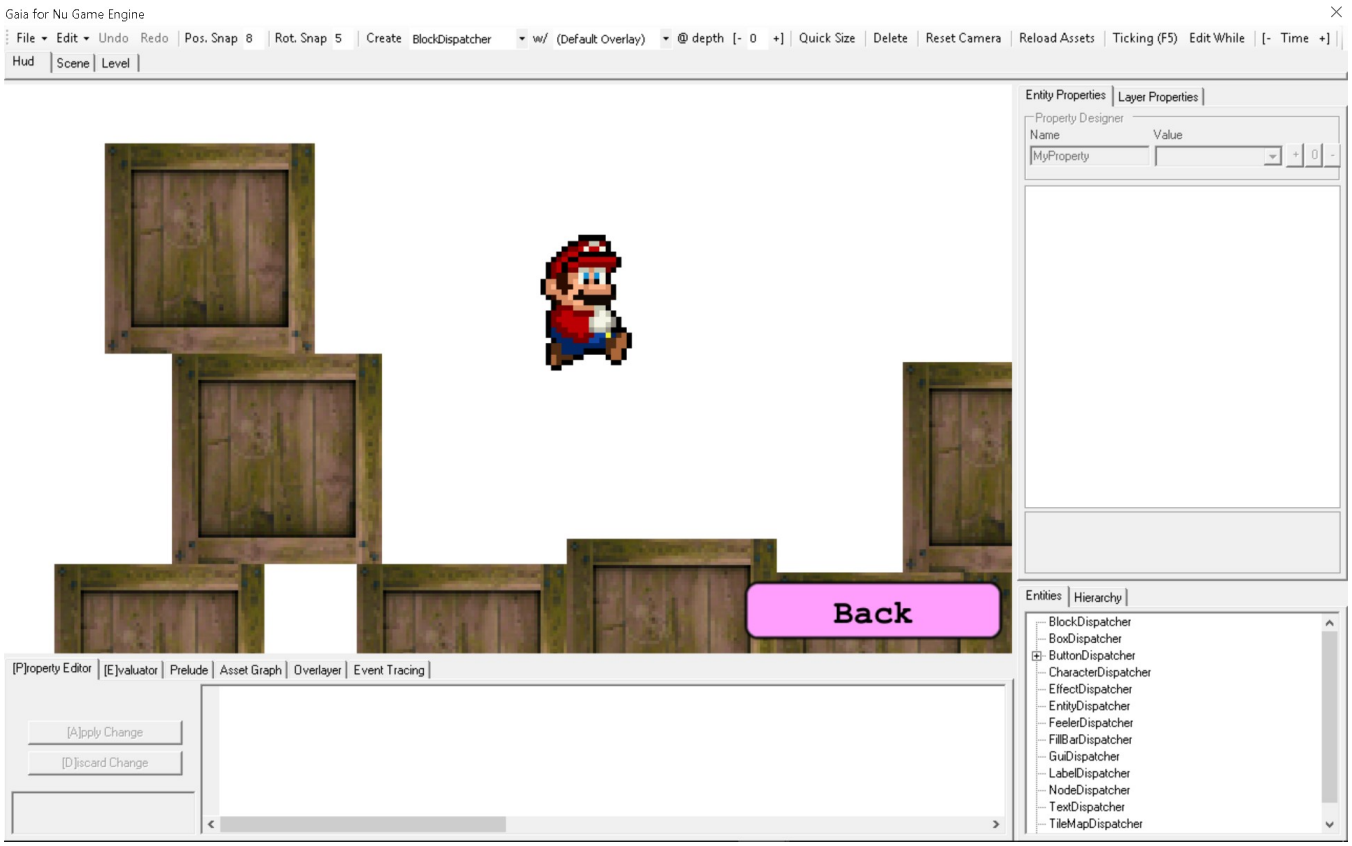
Shut down the editor when you're done.

Loading Your Game in Gaia

Next, lets load up our newly-created game in the editor! A nice feature of Nu is that you can play your game directly in the editor while editing. Before starting, make sure your game is built. If you want to make sure it's always up to date before running Gaia, add it as a **Build Dependency** to the **Nu.Gaia** project. Once's we're ready start a new session of Gaia and fill out the start dialog like so -



You will see the game loaded in the editor like so -



By selecting the tab of a containing Layer at the top left, you can edit that Layer's child simulants. So if you want to select and edit the Back button, have the **Hud** tab selected. To edit the Player character, select the **Scene** tab.

BlazeVector Sample Game

For a more in-depth example to study, check out the **BlazeVector** game project.

The Game Engine

By looking at the initial example, you might be able to make a vague inference 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] --> [Layer] --> [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 **Layers** 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 box with physics and a Gui button – they are both build from the same abstractions.

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 event 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 another World value.

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 above example -

```
override dispatcher.Command (command, _, _, world) =
    match command with
    | ShowTitle -> World.transitionScreen Simulants.Title world
    | ShowCredits -> World.transitionScreen Simulants.Credits world
    | ShowGameplay -> World.transitionScreen Simulants.Gameplay world
    | ExitGame -> World.exit world
```

Layer

Layers represent logical collections of entities that can be combined to make up a Screen. Each layer has a Depth property that offsets the depth of all its entities at run-time. However, that does not mean all entities in a given layer will be above or below all the entities in another layer. Multiple layers can be side-by-side by leaving their Depth properties to the default of 0.

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

Simulant Handles

Simulants are not accessed and transformed directly, but rather through handle types such as Entity, Layer, Screen, and Game. Simulant handles are created from addresses that uniquely identify a given simulant - EG, **let entity = Entity entityAddress**. We'll elaborate more on addresses next.

Addresses

You may be wondering how the engine locates specific entities as created in Gaia and loaded from the saved ***.nulyr** 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/LayerName/EntityName**, where **ScreenName** is the name that is given to its containing screen, **LayerName** is the name given to its containing layer, and **EntityName** is the name given to the entity (such as in the editor). Remember how we changed the **Name** property of the button object that we created to **MyButton** earlier in **Gaia**? That's the **EntityName** portion of its address! The same structure applies to screen and layers 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 **Events.MouseMove** has a generic type of **MouseMoveData**, and an **EntityAddress** has a generic type of **Entity**. Additionally, there are several operators and conversion functions used to combine addresses and manipulate their type appropriately in the **Address.fs** and **SimulationOperators.fs** files of the **Nu** project. With these functions, you can combine simulant addresses with the common event address value found in **SimulationEvents.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 had to invent a custom, purely-functional event system for Nu.

Subscriptions are created by invoking the **World.subscribe** function, and destroyed using the **World.unsubscribe** 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.

I won't cover this in too much detail since, for the most part, you'll be using the Elm-style bindings to wire up your events for you.

Xtensions

Xtensions are a key enabling technology in Nu. Xtensions allow the **Game**, **Screen**, **Layer**, 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>] canAddProperty () =
    let xtn = Xtension.empty
    let xtn = xtn?TestProperty <- 5
    let propertyValue = xtn?TestProperty
    Assert.Equal (5, propertyValue)
```

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 property called **TestProperty** 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 properties (the 'empty' Xtension). By using the **dynamic (?<-) operator** as shown on the third line, **xtn** is augmented with a property named **TestProperty** that has a value of **5**. The next line then utilizes the **dynamic (?) operator** to retrieve the value of the newly added property into the **propertyValue** variable. Note the surprising presence of strong typing on the **propertyValue** 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?TestProperty <- 5
    let propertyValue = xtn?TestProperty
    propertyValue
```

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?TestProperty <- 5
    let propertyValue = xtn?TestProperty : int
    propertyValue
```

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 properties 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 = this.Get Property? Density
    member this.SetDensity = this.Set Property? Density
    member this.Density = Lens.make<single, World> Property? Density this.GetDensity this.SetDensity this
```

- which, when used in practice, looks 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 properties are, in actuality, dynamic!

You'll also notice the member **Density** of type **Lens**. Each property should be accompanied by a related lens in order for it to participate in Nu's iterative functional reactive programming model. The lens is used to specify the initial value of the property, construct change events, among other things.

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 hearkens 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 property in the Entity type defined as –

```
    DispatcherNp : EntityDispatcher
```

This is the property 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 Properties =
        [Define? UserState (UserState.make () false)
         Define? Persistent true
         Define? Position Vector2.Zero
         Define? Size Constants.Engine.DefaultEntitySize
         Define? Rotation 0.0f
         Define? Depth 0.0f
         Define? Overflow Vector2.Zero
         Define? ViewType Relative
         Define? Visible true
         Define? Enabled true
         Define? Omnipresent false
         Define? AlwaysUpdate false
         Define? PublishChanges true
         Define? PublishUpdatesNp false
         Define? PublishPostUpdatesNp false
         Define? Persistent true]

    /// Register an entity when adding it to a layer.
    abstract Register : Entity * World -> World
    default dispatcher.Register (_, world) = world

    /// Unregister an entity when removing it from a layer.
    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

    /// Update an entity.
    abstract Update : Entity * World -> World
    default dispatcher.Update (_, world) = world

    /// Actualize an entity.
    abstract Actualize : Entity * World -> World
    default dispatcher.Actualize (_, world) = world

    /// 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

    /// Try to get a calculated property with the given name.
    abstract TryGetCalculatedProperty : string * Entity * World -> Property option
    default dispatcher.TryGetCalculatedProperty (_, _, _) = None
```

The **Properties** member is 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. **Update** is your typical update callback. **Actualize** is what can 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 Gaia. **TryGetCalculatedProperty** exposes purely computed user-defined properties to the editor and scripting system.

All these overrides are available for you to customize your entity's behavior. But that's not the only way. You can instead use the generic **EntityDispatcher<_, _, _>** type to implement your entity using the Elm-style with its available overrides -

```
abstract member Bindings : 'model * Entity * World -> Binding<'message, 'command, Entity, World> list
default this.Bindings (_, _, _) = []

abstract member Message : 'message * 'model * Entity * World -> 'model * 'command list
default this.Message (_, model, _, _) = just model

abstract member Command : 'command * 'model * Entity * World -> World
default this.Command (_, _, _, world) = world

abstract member Content : Lens<'model, World> * Entity * World -> EntityContent list
default this.Content (_, _, _) = []

abstract member View : 'model * Entity * World -> View list
default this.View (_, _, _) = []
```

Generally, the Elm-style of implementing entity is recommended unless you have some reason to use the lower-level style.

Facets

Don't we need some form a composition in order to reuse behaviors among different entities?

Of course, and that's what **Facets** are for!

A F**acet** 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 look at the definition and use of one of Nu's most basic facets now –

```
[<AutoOpen>]
module StaticSpriteFacetModule =

    type Entity with

        member this.GetStaticImage world : Image AssetTag = this.Get Property? StaticImage world
        member this.SetStaticImage (value : Image AssetTag) world = this.Set Property? StaticImage value world
        member this.StaticImage = Lens.make Property? StaticImage this.GetStaticImage this.SetStaticImage this

    type StaticSpriteFacet () =
        inherit Facet ()

        static member Properties =
            [define Entity.StaticImage { PackageName = Assets.DefaultPackageName; AssetName = "Image3" }]

        override facet.Actualize (entity, world) =
            if entity.InView world then
                World.addRenderMessage
                    (RenderDescriptorsMessage
                        [|LayerableDescriptor
                            { Depth = entity.GetDepth world
                              LayeredDescriptor =
                                SpriteDescriptor
                                    { Position = entity.GetPosition world
                                      Size = entity.GetSize world
                                      Rotation = entity.GetRotation world
                                      ViewType = entity.GetViewType world
                                      InsetOpt = None
                                      Image = entity.GetStaticImage world
                                      Color = Vector4.One }}|])
                    world
            else world

        override facet.GetQuickSize (entity, world) =
            match Metadata.tryGetTextureSizeAsVector2 (entity.GetStaticImage world) world.State.AssetMetadataMap with
            | Some size -> size
            | None -> Constants.Engine.DefaultEntitySize
```

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

Similar to the StaticSpriteFacet, 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 this.GetMinorId world : Guid = this.Get Property? MinorId world
        member this.SetMinorId (value : Guid) world = this.SetFast Property? MinorId false false value world
        member this.MinorId = Lens.make Property? MinorId this.GetMinorId this.SetMinorId this
        member this.GetBodyType world : BodyType = this.Get Property? BodyType world
        member this.SetBodyType (value : BodyType) world = this.SetFast Property? BodyType false false value world
        member this.BodyType = Lens.make Property? BodyType this.GetBodyType this.SetBodyType this
        member this.GetAwake world : bool = this.Get Property? Awake world
        member this.SetAwake (value : bool) world = this.SetFast Property? Awake false false value world
        member this.Awake = Lens.make Property? Awake this.GetAwake this.SetAwake this
        member this.GetDensity world : single = this.Get Property? Density world
        member this.SetDensity (value : single) world = this.SetFast Property? Density false false value world
        member this.Density = Lens.make Property? Density this.GetDensity this.SetDensity this
        member this.GetFriction world : single = this.Get Property? Friction world
        member this.SetFriction (value : single) world = this.SetFast Property? Friction false false value world
        member this.Friction = Lens.make Property? Friction this.GetFriction this.SetFriction this
        // remaining physics property definitions elided for space...

    type RigidBodyFacet () =
        inherit Facet ()

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

        static member Properties =
            [variable Entity.MinorId ^ fun () -> Core.makeId ()
             define Entity.BodyType Dynamic
             define Entity.Awake true
```

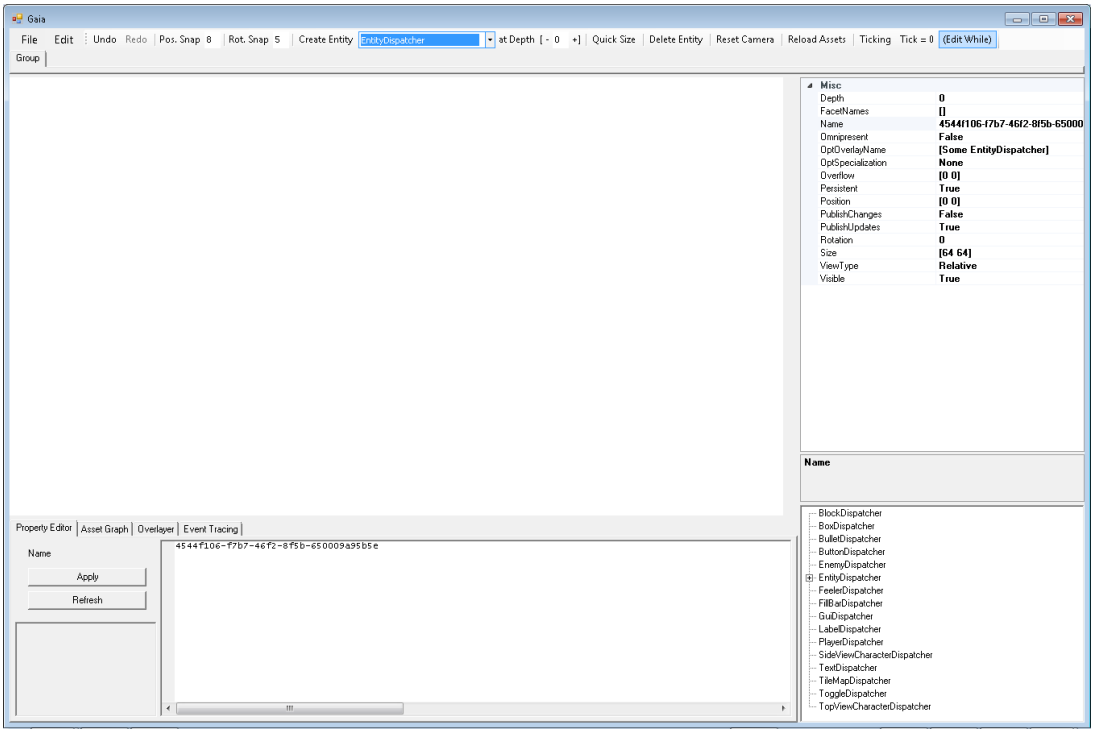
```
            define Entity.Density Constants.Physics.NormalDensity
            define Entity.Friction 0.0f
            define Entity.Restitution 0.0f
            define Entity.FixedRotation false
            define Entity.AngularVelocity 0.0f
            define Entity.AngularDamping 1.0f
            define Entity.LinearVelocity Vector2.Zero
            define Entity.LinearDamping 1.0f
            define Entity.GravityScale 1.0f
            define Entity.CollisionCategories "1"
            define Entity.CollisionMask "*"
            define Entity.CollisionExpr "[BoxShape [[0.5 0.5] [0.0 0.0]]]"
            define Entity.IsBullet false
            define Entity.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
                  Awake = entity.GetAwake world
                  Density = entity.GetDensity world
                  Friction = entity.GetFriction world
                  Restitution = entity.GetRestitution world
                  FixedRotation = entity.GetFixedRotation world
                  AngularVelocity = entity.GetAngularVelocity world
                  AngularDamping = entity.GetAngularDamping world
                  LinearVelocity = entity.GetLinearVelocity world
                  LinearDamping = entity.GetLinearDamping 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 –

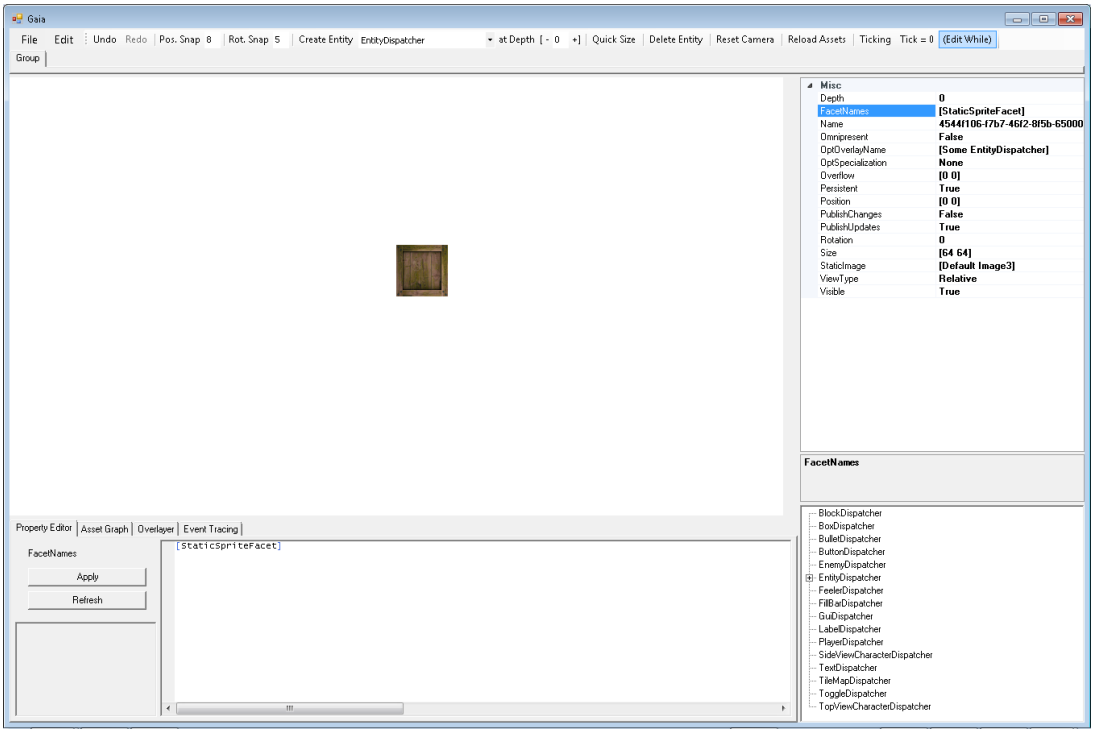
```
[<AutoOpen>]
module RigidSpriteDispatcherModule =

    type RigidSpriteDispatcher () =
        inherit EntityDispatcher ()

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

Additionally, facets can be dynamically added to a removed from an entity in Gaia 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 –


Gaia
File
Edit
Undo
Redo
Pos. Snap 8
Rot. Snap 5
Create Entity
EntityDispatcher
at Depth [- 0 +]
Quick Size
Delete Entity
Reset Camera
Reload Assets
Ticking
Tick = 0
(Edit While)
Group
Misc
Depth 0
FacetNames []
Name 4544f106-f7b7-46f2-8f5b-65000
Omnipresent False
OptOverlayName [Some EntityDispatcher]
OptSpecialization None
Overflow [0 0]
Persistent True
Position [0 0]
PublishChanges False
PublishUpdates True
Rotation 0
Size [64 64]
ViewType Relative
Visible True
Name
Property Editor
Asset Graph
Overlayer
Event Tracing
Name
Apply
Refresh
4544f106-f7b7-46f2-8f5b-650009a95b5e
BlockDispatcher
BoxDispatcher
BulletDispatcher
ButtonDispatcher
EnemyDispatcher
EntityDispatcher
FeelerDispatcher
FillBarDispatcher
GuiDispatcher
LabelDispatcher
PlayerDispatcher
SideViewCharacterDispatcher
TextDispatcher
TileMapDispatcher
ToggleDispatcher
TopViewCharacterDispatcher

Notice how nothing 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** property to **[StaticSpriteFacet]** –



Not only does it now render, the additional properties 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 **[RigidBodyFacet StaticSpriteFacet]**, and then toggling on the **Ticking** button.

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

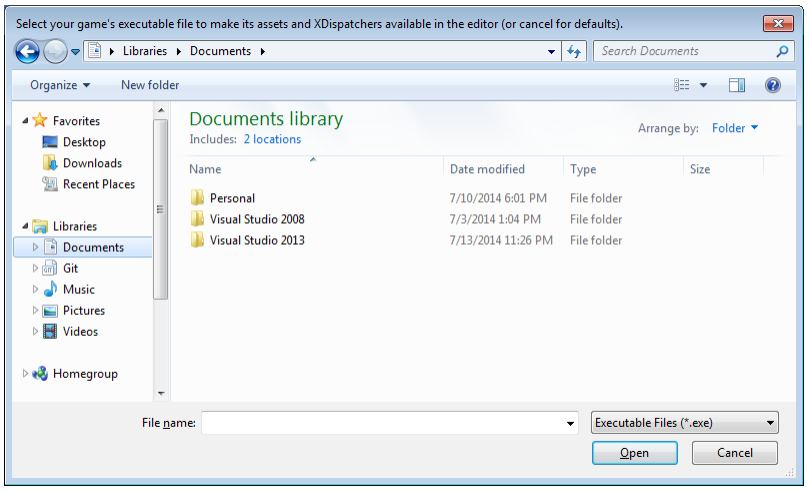
Now that we know more about the Nu Game Engine, we can explore more deeply the implementation of **BlazeVector**. In this section, we'll be loading up some of the entities used in BlazeVector in Gaia so that we can interact with each in isolation. We'll also use that interaction as a chance to study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

Bullets and the BulletDispatcher

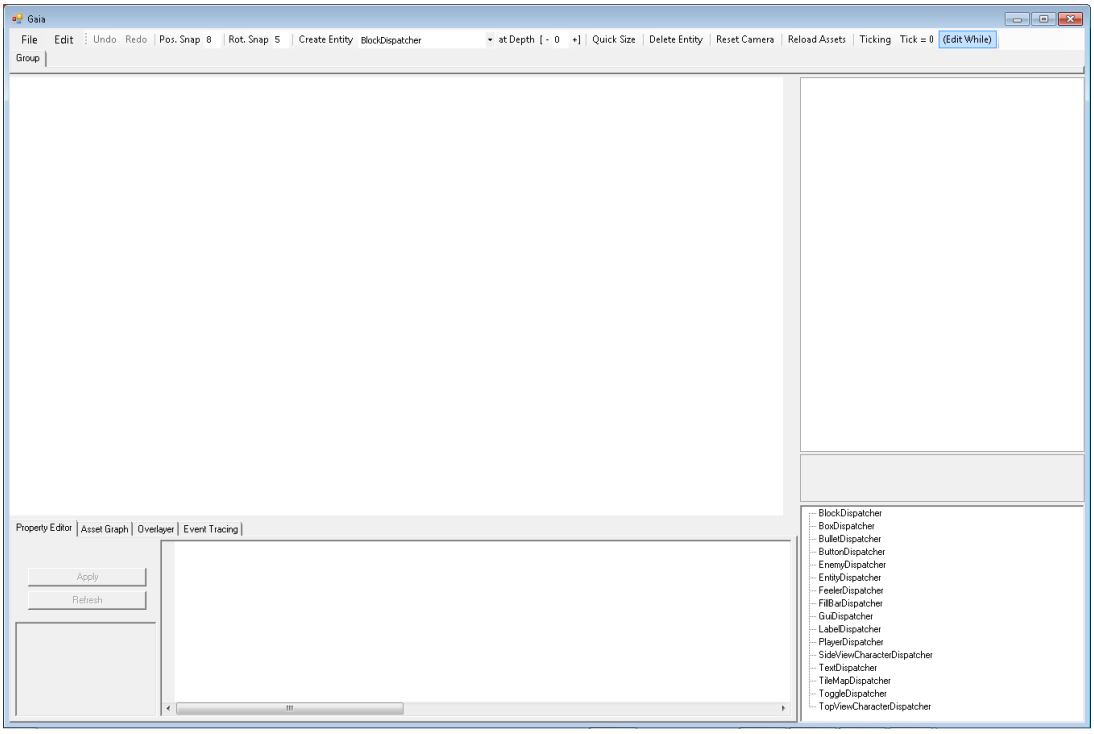
We wouldn't have much of a shooting game in BlazeVector if we didn't have bullets! Since bullets are the simplest entities defined in the BlazeVector project, let's study them first.

Bullets in Gaia

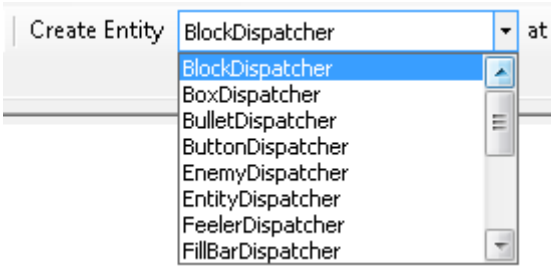
First, we'll play with a few bullet entities in the editor. If it's not already open, once again open the Nu.sln, set the Gaia project as the **StartUp** project, and then run it. As you know, you will see an Open File dialog appear like so –



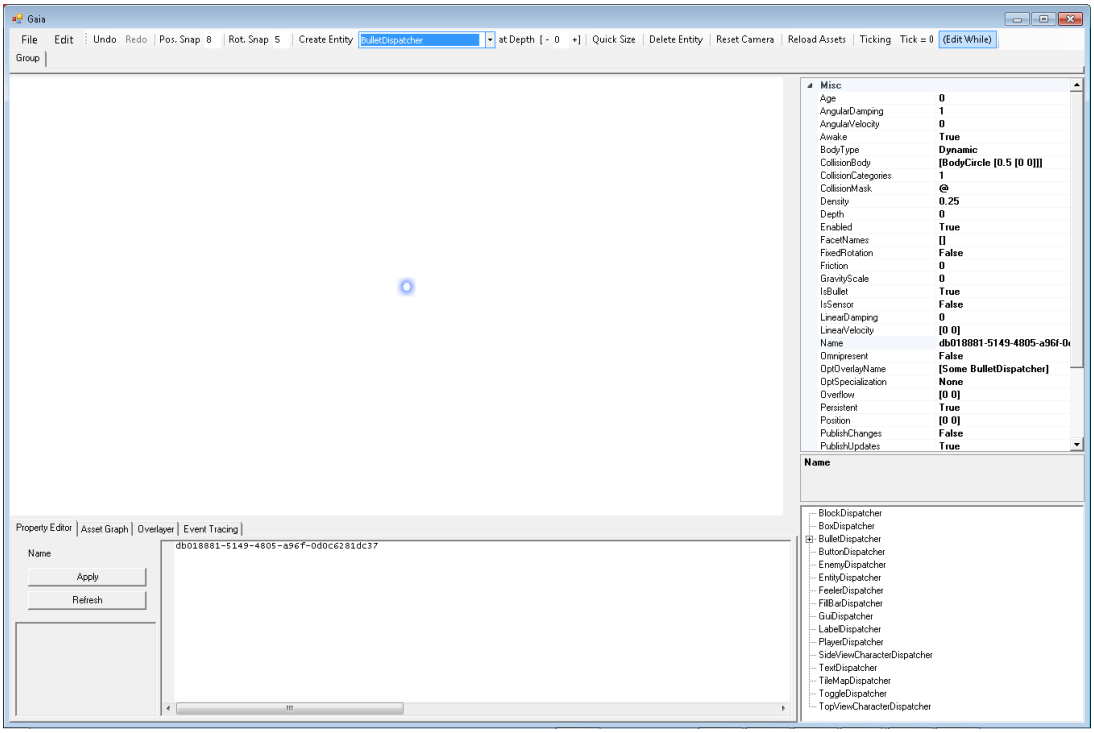
Since we need the **BulletDispatcher** from the BlazeVector.exe file, navigate the Open File dialog to the **./Projects/BlazeVector/ bin/Debug** folder and select the **BlazeVector.exe** file. The editor will now open up as normal –



- 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 –

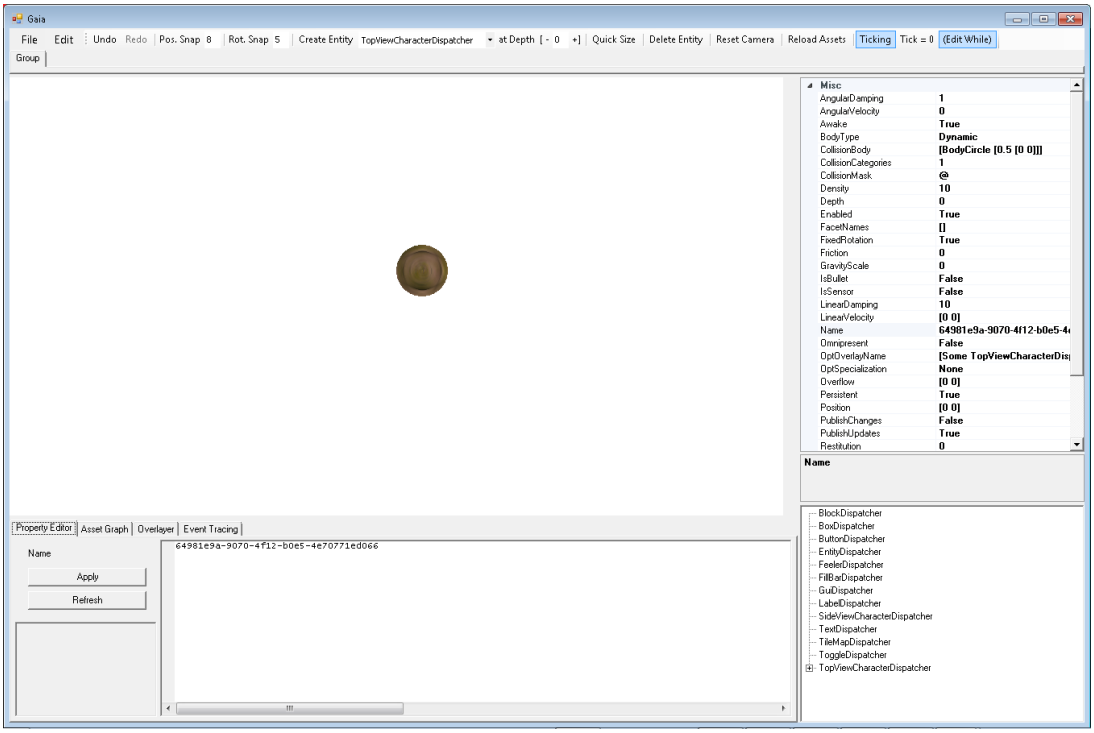


The bullet doesn't really have much behavior, but that's because the **Ticking** button is not toggled on. Let's try toggling it on now...

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 **Ticking** toggled 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 –



Now let's select **BulletDispatcher** again, and create another one.

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. Next, trying moving the character so that the bullet isn't created on top of him, and then creating a bullet. It sticks around for its given lifetime, then disappears.

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 = this.Get Property? Age
        member this.SetAge = this.Set Property? Age
        member this.Age = Lens.make<int64, World> Property? Age this.GetAge this.SetAge this

    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 FacetNames =
        [typeof<RigidBodyFacet>.Name
         typeof<SpriteFacet>.Name]

    static member Properties =
        [define Entity.Size (Vector2 (24.0f, 24.0f))
         define Entity.Density 0.25f
         define Entity.Restitution 0.5f
         define Entity.LinearDamping 0.0f
         define Entity.GravityScale 0.0f
         define Entity.IsBullet true
         define Entity.CollisionExpr "[CircleShape [0.5 [0.0 0.0]]]"
         define Entity.SpriteImage Assets.PlayerBulletImage
         define Entity.Age 0L]

    override dispatcher.Register (bullet, world) =
        let world = World.monitor handleUpdate Events.Update bullet world
        World.monitor handleCollision bullet.CollisionEvent 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 Nu Game Engine repository.

```
    type Entity with

        member this.GetAge = this.Get Property? Age
        member this.SetAge = this.Set Property? Age
        member this.Age = Lens.make<int64, World> Property? Age this.GetAge this.SetAge this
```

If you recall back to the **The Xtension System** section, you'll understand that a new property **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 Properties =
            [define Entity.Size (Vector2 (24.0f, 24.0f))
             define Entity.Density 0.25f
             define Entity.Restitution 0.5f
             define Entity.LinearDamping 0.0f
             define Entity.GravityScale 0.0f
             define Entity.IsBullet true
             define Entity.CollisionExpr "[CircleShape [0.5 [0.0 0.0]]]"
             define Entity.SpriteImage Assets.PlayerBulletImage
             define Entity.Age 0L]
```

You might be wondering how dispatchers and facets specify what properties are added to entities. What you see before you is how! By specifying the static **Properties** property for a given dispatcher or facet like so, the engine will automatically attach the defined properties with the corresponding values to the entity at run-time
The above definitions define the bullet's properties to give it physical and rendering properties becoming of a bullet, as well as initialize the user-defined **Age** property to 0.

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

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

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

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 **PropertyNames** property 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 bullet.UpdateEvent bullet |>
        World.monitor handleCollision bullet.CollisionEvent 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 **handleUpdate** 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 function

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 from the **event.Subscriber** property, denoting to the type system that it is an entity with the type annotation at the end.

The next line increments the bullet’s **Age** property 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 function

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)
```

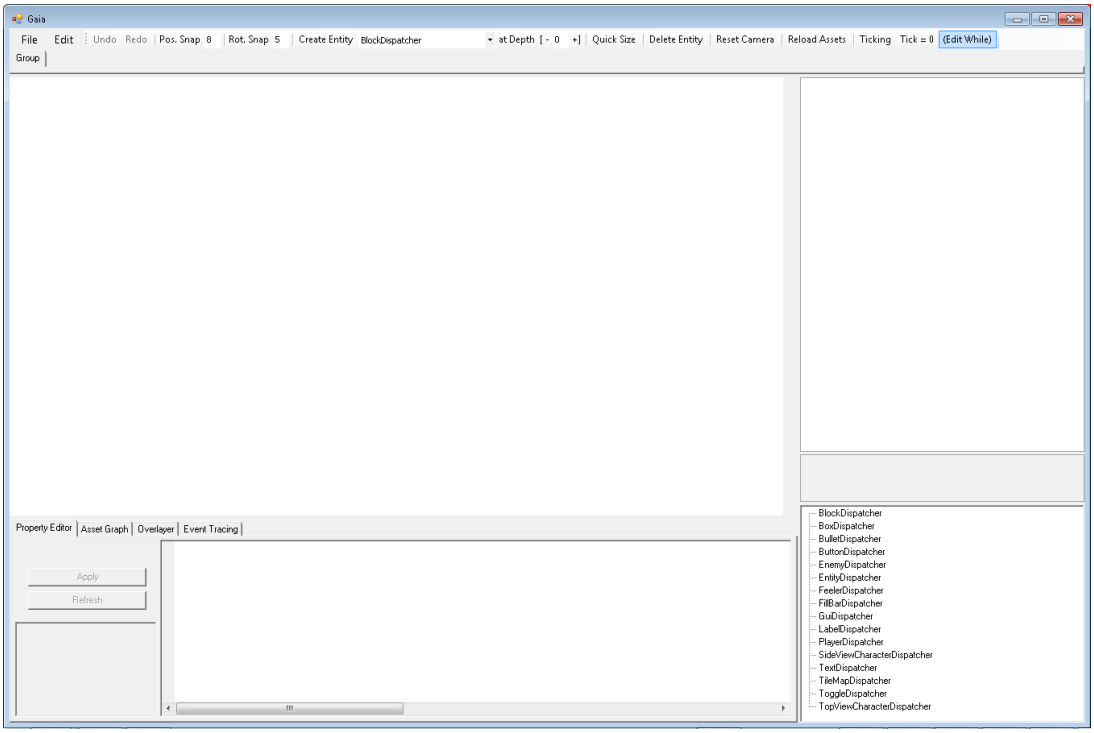
Even simpler than the previous handler, it merely destroys the bullet when a collision with it takes place while the world is ticking (EG – the world's TickRate is not zero).

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.

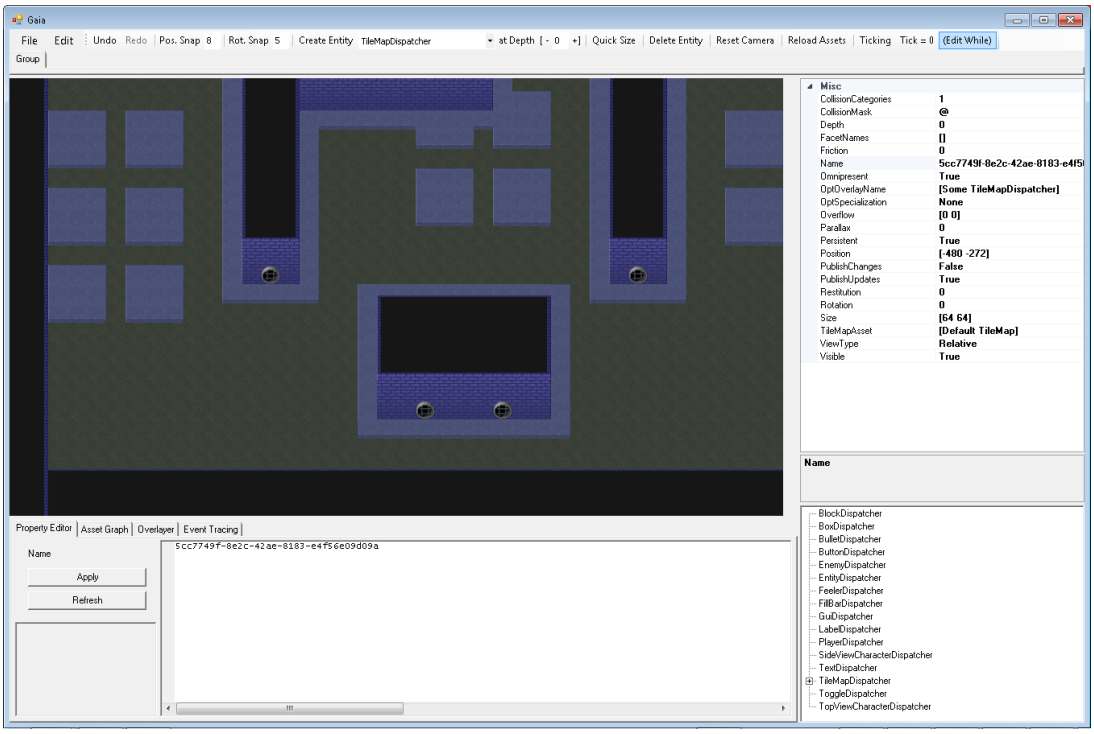
Enemies in Gaia

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



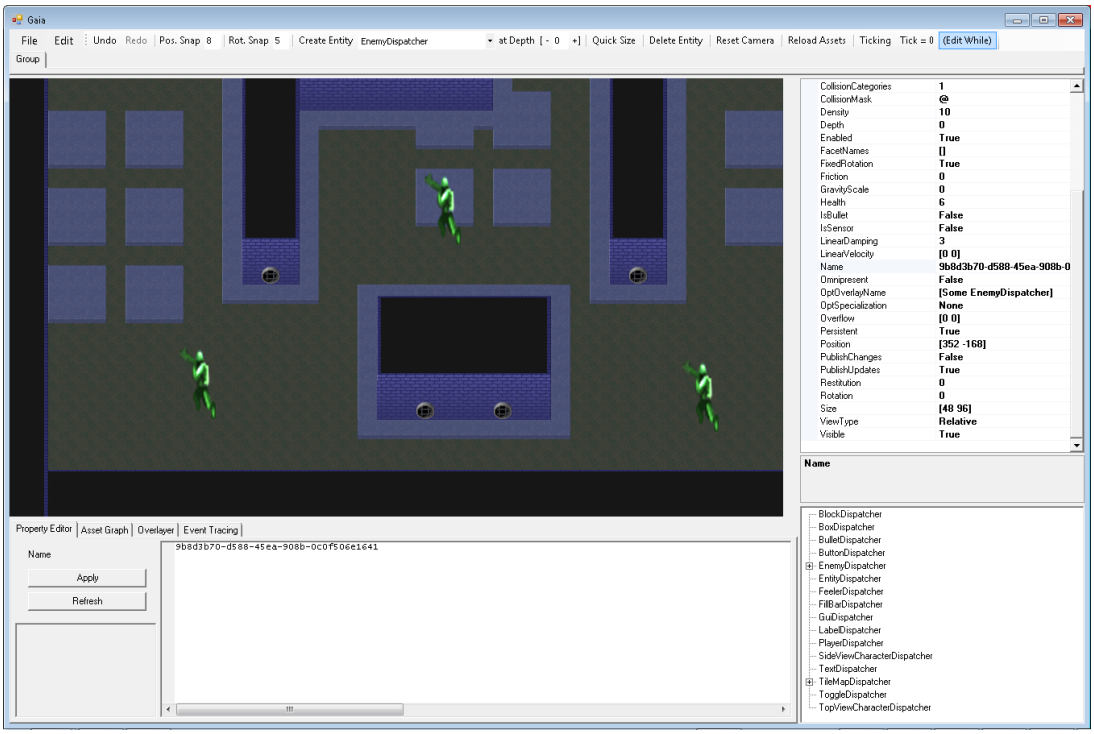
Before creating enemies, let's create a tile map in which for them to live by selecting the **TileMapDispatcher**, clicking **Create Entity**, and then dragging the bottom-left corner of the tile map to the bottom left corner of the screen

We will end up with something like this –

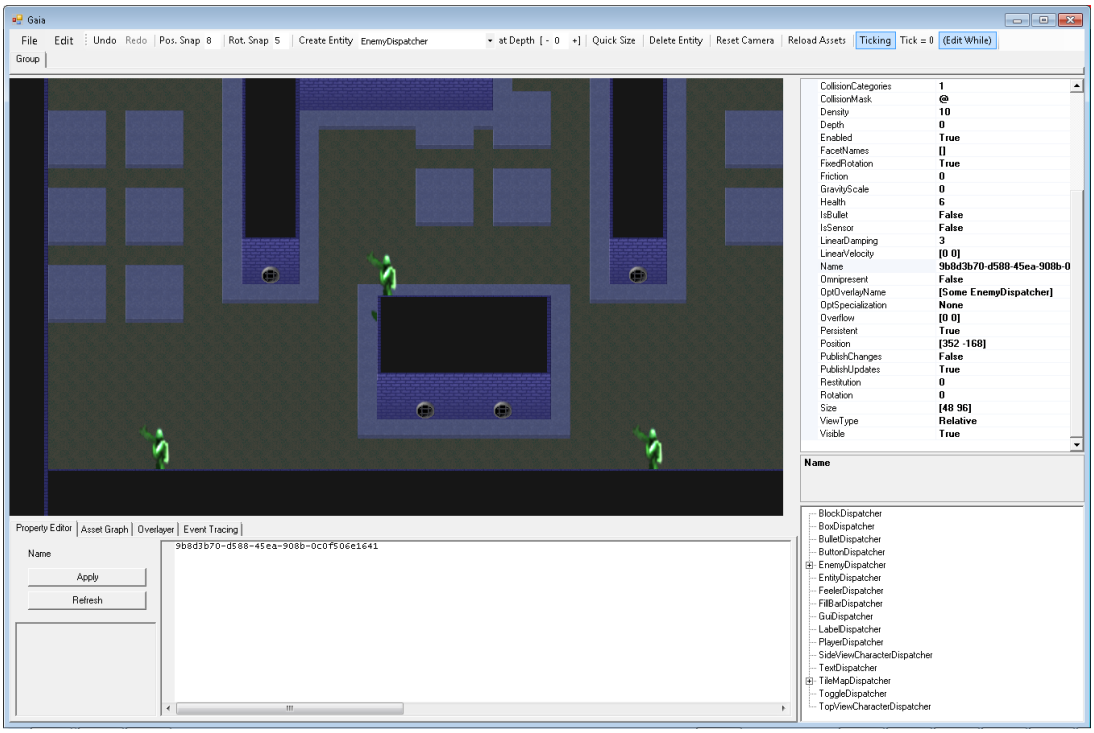


Since we want to create additional entities on top of the bottom layer of the tile map (but below the second layer), we set **at Depth** value in the editor's tool bar to **1**. Next, we create a few enemies by selecting the **EnemyDispatcher** and clicking **Create Entity** a few times (or alternatively, 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 a file named **AssetGraph.nuag**. 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 **Nu.Pipe.exe**. Nu.Pipe knows what assets to build by itself consulting the game's Asset Graph. During the build process of a given Nu game project, Nu.Pipe is invoked from the build command line like so –

"$(ProjectDir)..\..\Nu\Nu.Pipe\bin\$(ConfigurationName)\Nu.Pipe.exe" "$(ProjectDir)\" "$(TargetDir)\" "$(ProjectDir)refinement" False

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

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

```
[[Default
  [[Asset Font "Assets/Default/FreeMonoBold.032.ttf" [Render] []]
   [Assets Assets/Default nueffect [Symbol] []]
   [Assets Assets/Default nuscript [Symbol] []]
   [Assets Assets/Default csv [Symbol] []]
   [Assets Assets/Default png [Render] []]
   [Assets Assets/Default wav [Audio] []]
   [Assets Assets/Default ogg [Audio] []]
   [Assets Assets/Default tmx [] [][]]]]
```

This file uses Nu's s-expression syntax. There is a single **Package** that holds multiple **Asset** descriptors. 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 property 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 Gaia, 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.nuag). 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 Gaia where the game's currently loaded assets can be rebuilt and reloaded at run-time. This is done by pressing the **Reload Assets** button found at the top-right of the window.

Serialization

By default, all of your simulation types can be serialized at any time to a 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 property from being serialized, simply end its name with the letters '**Np**'. This suffix stands denotes two things about a property; it is **N**on-**P**ublishing (it never raises events when changed), and it is **N**on-**P**ersistent (doesn't serialize). Also keep in mind its opposing suffix '**Ap**', which just stands for **A**lways-**P**ublishing.

When you save a scene in Gaia, you may notice that not all of a given entity's properties at actually written out, even if they don't end with '**Np**'. This is helps keep files small and quicker to load, 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 property values that multiple entities hold in common. Overlays are defined in a file that is included with every new Nu game project called **Overlayer.nuol**. 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 **Properties**.

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

```
[[SampleOverlay
  [BlockDispatcher]
  [[BodyType Dynamic]
   [Friction 0.5]
   [IsSensor True]]]]
```

Where overlays get interesting is when they are applied to an entity at run-time. Say you're in Gaia and you want to have a common set of button property 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 properties 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 Overlayer.nuol file (ensuring the specified click sound asset exists and is also declared in the AssetGraph.nuag file) –

```
[[MyButtonDispatcher [ButtonDispatcher]
  [[ClickSoundOpt [Some [Gui Affirm]]]]]]
```

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

Third, for each button you wish to overlay, change its **OverlayNameOpt** property to **[Some MyButtonOverlay]**.

Voila! Both the **Enabled** and **ClickSoundOpt** properties 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 properties to something other than what was specified in its previous overlay! You see, overlay values are applied to only to the properties 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 properties 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 names in the Overlayer.nuol file (like is done with **[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).

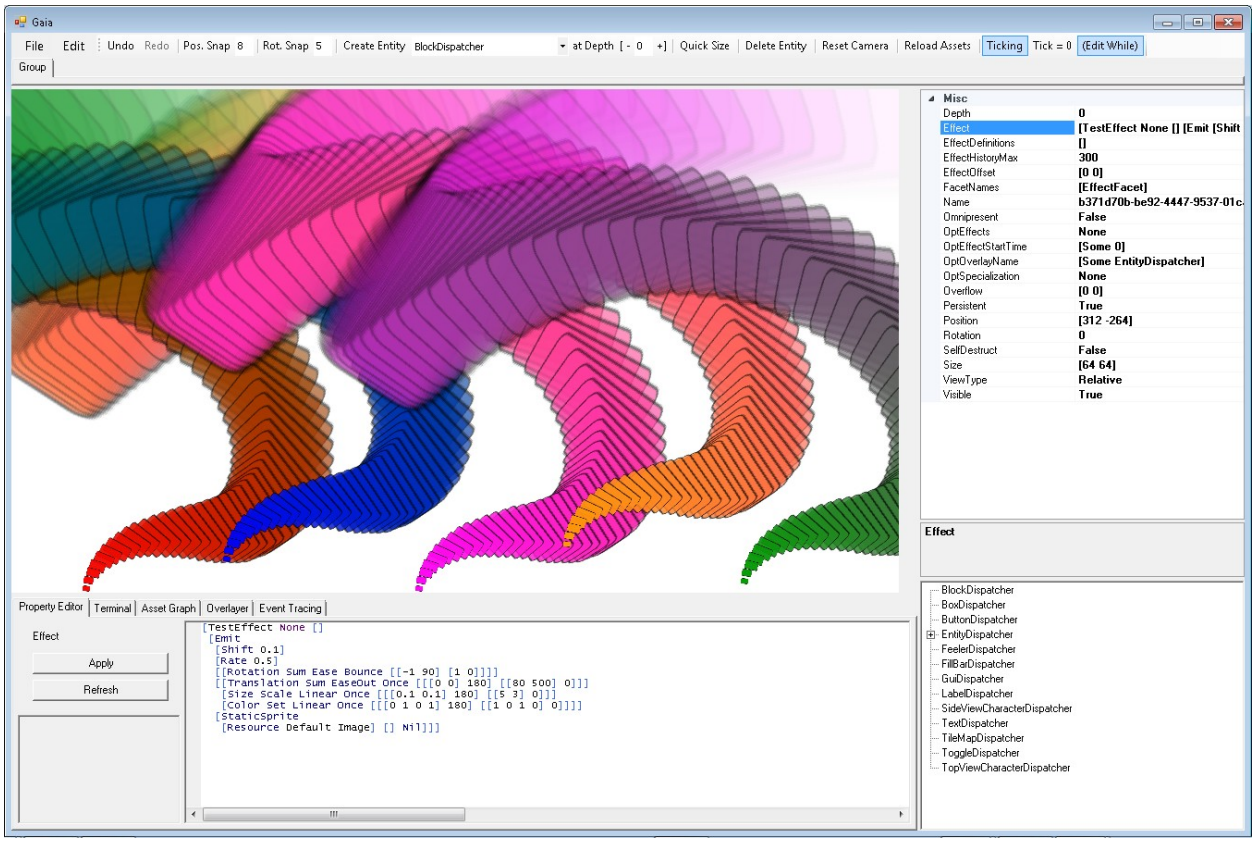
Now, of course the use of message queues can make accomplishing certain things a little more complicated due to the inherent indirection it entails. Not only is the call-site a bit separated from the target, the time at which the actual message is handled is also separated. These two facts can make debugging 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** or a high-performance **Particle System** can be added by overriding the **MakeSubsystems** method in your **NuPlugin** subtype.

Special Effects System

A recent addition to the Nu Game Engine is the special effects system called **EffectSystem**.



To apply an effect to an Entity, you add the **EffectFacet** to its **FacetNames** property, and modify its **Effect** property to specify the desired effect using the effect syntax described below.

Effect Syntax

It is composed of a DSL using symbolic expressions, and a short list of composable semantics –

[Expand *definitionName* [*args…*]] – Expand a Content definition (more on definitions later).

[StaticSprite *resource* [*aspects…*] *childContent*] – Display a static sprite with the given Aspects (more on aspects later) and optional mounted mouted Content.

[AnimatedSprite *resource celSize celRun celCount stutter* [*apsects…*] *childContent*] – Display an animated sprite with the given properties, Aspects, and optional mounted child Content.

[Mount [Shift *shift*] [*aspects…*] *childContent*] – Mount the given child content with the given depth shift amount and Aspects.

[Repeat [Shift *shift*] ([Iterate *iterations*] | [Cycle *cycles*]) [*incrementAspects*] *childContent*] – Repeatedly invoke the given child Content with the given number of iterations or cycles of the given increment Aspects. Intuitively, like a declarative 'for' loop with either the incrementAspects applied iteratively or in a cycle.

[Emit [Shift *shift*] [Rate *rate*] [*emitterAspects…*] [*aspects…*] *childContent*] – Emit the given child Content at the given rate with the given Aspects. Note that due to performance limitations, Emit does not inherit its aspects from its parent, but has its own emitter Aspects.

Note also that emitters don't yet stack on other emitters due to a lack of implementation. This is coming soon, however!

[Composite [Shift *shift*] [*childContents…*]] – Compose multiple child Contents.

[Tag *name* `*quotedValue*'] – Tags an effect with given name paired with the given quoted value. Tags can be pulled from an Entity's EffectTags map at run-time via the given name, and have the quoted values observed or evaluated separately.

Nil – Specifies a lack of further Content.

As you might notice, the DSL syntax is built purely out of Nu's symbolic serialization syntax (or, symbolic-expressions (or *s-exprs* for short)). So to understand how the syntax operates in full, you simply need to reference the structure of the effect data types in the 'Nu/Nu/Nu/Effect.fs' source file.

Effect Aspects

In addition to normal parameters, Effect semantics allow modification via **Aspects**. Aspects specify the **Position**, **Size**, **Color**, and other properties of an Effect which can be animated over multiple **Key Frames** and inherited via implicit or explicit **Mounting**.

First, let's cover the general syntax of Aspects –

[*AspectName logic algorithm playback* [*keyFrames…*]] – The AspectName is, well, the name of the Aspect that is to be modified. Possibilities here include -

The logic value is any one of the following: **Or | Nor | Xor | And | Nand | Equal**. The logic value applies the value from the Aspect's current animation frame to its inherited property. So if the Enabled aspect is True on the current frame and the logic value is set to **And**, then the resulting Enabled property will be True if and only if the inherited Enabled value is True.

The remaining possible logic values are self-explanatory, except maybe for **Eq**. Eq simply takes the current frame value and ignores the inherited value.

The algorithm value exists on non-boolean Aspects. It is any one of the following: **Const | Linear | Random | Chaos | Ease | EaseIn | EaseOut | Sin | Cos**.

The playback value is any one of the following: **Once | Loop | Bounce**. Once means that the animation will play once and then stop at the last Key Frame. Loop means the animation will play repeatedly from start to finish. Bounce means the animation will be played alternatively forward and backward.

Finally, of note, are the key frames, which have their own syntax like the following –

[[True 0] [False 10] [True 0]]

As you can see, it is just a list of Boolean values with the number of frames for which the value should hold. For the above, the value will be True on the first frame, False for 10 frames thereafter, then True after that. You can

have as many Key Frames as you like.

Here are some example Key Frames for the Position aspect –

[**[[0 0] 10] [[100 100] 170]]**

It's a little more verbose since Vector2s need to be in their own list.

Here is the syntax for each Aspect in detail -

[Enabled *logic playback* [*keyFrames…*]] This property dictates whether an effect is enabled – which has a different meaning depending on the Content. For example, if the Content is StaticSprite, it dictates if the StaticSprite is displayed that frame. If a SoundEffect, it dictates if the sound effect is to be played that frame. If an Emit effect, it dictates if the emitter is emitting that frame.

TODO: rest of aspects!!!

Proper Effect Rendering with Entity Overflow

If you pan the screen off of an entity that's outputting an effect outside of its normal bounds, you may see the effect disappear unexpectedly. This is because Nu's culling system thinks the effect need not be processed due to the entity being out of culling bounds.

Therefore, it is important to understand the **Overflow** property of entities, and to adjust them properly.

The **Overflow** property of each entity expands the bounds of each entity by a multiple of its value. So if its value is [0 0] (which is the default), no bounds expansion happens. If it is [1 1], then the bounds are expanded by 100% of the original size, and so on.

So you must estimate the appropriate **Overflow** for entities with effects in order to adjust their bounds for proper culling.

Sample Effects

TODO