Nu Game Engine

The world's first practical functional game engine!

What's It All About?

The Nu Game Engine is a **Mature**, **Functional**, **2d (and soon 3D!) Game Engine** written in **F#**.

Let me explain each of those terms –

Mature

Nu is mature and exposes multiple levels of programmability depending on your performance needs. At the high level, there's the Elm-style programming API. At the low level, there's an Entity-Component-System API called **ECS**. By default you'll be using the Elm-style API, but you can drop down to the ECS when you need to scale to hundreds of thousands of entities

Additionally, there is a tile map system that utilizes **Tiled#**, and there is a physics system that utilizes **Aether Physics**. 2D and Physically-Based 3D rendering is done with **OpenGL**. Audio and other IO systems are handled in a cross-platform way with **SDL2** / **SDL2#**. There is also an asset management system to make sure your game can run on memory-constrained devices such as the iPhone. There is a declarative special effects system called, appropriately enough, **EffectSystem**, as well as an efficient **ParticleSystem**. On top of all that, there is a built-in game editor called **Gaia**!

Functional / Imperative Configuration

Nu can be configured to run in an immutable mode. This is how the world editor, Gaia, implements Undo and Redo – by taking snapshots of the immutable world state. However, for optimal performance, Nu is configured to run with mutation under the hood outside of the editor. It can be configured either way in both context, however.

2D and 3D 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 and 3D* games. Nu is intended to be a broadly generic toolkit for 2D and 3D* game development.

**3D physics are currently being implemented for Nu, so thank you for your patience!*

F#

We know what F# is, so why use it? First, because of its **cross-platform** nature. Nu runs on both the .NET Framework and Windows and on Mono and Linux and other platforms. 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 an intuitive and a well-tooled debugging experience, with no need babysit effects / monads / ect. 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 at hand.

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 revision 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 Nu 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). **UPDATE:** ***I'm no longer sure if this is necessary! YMMV without it!***

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 games, **Nelmish** (a simple GUI programming example), **Elmario** (a simple physics game example), **BlazeVector** (a somewhat more sophisticated shooter game), and my WIP role-playing game **OmniBlade** (an actively developed commercial title). You should explore these projects in the order I've listed them here.

To open the Nu solution, first make sure to have **Visual Studio 2019** 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 **Nelmish** 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.

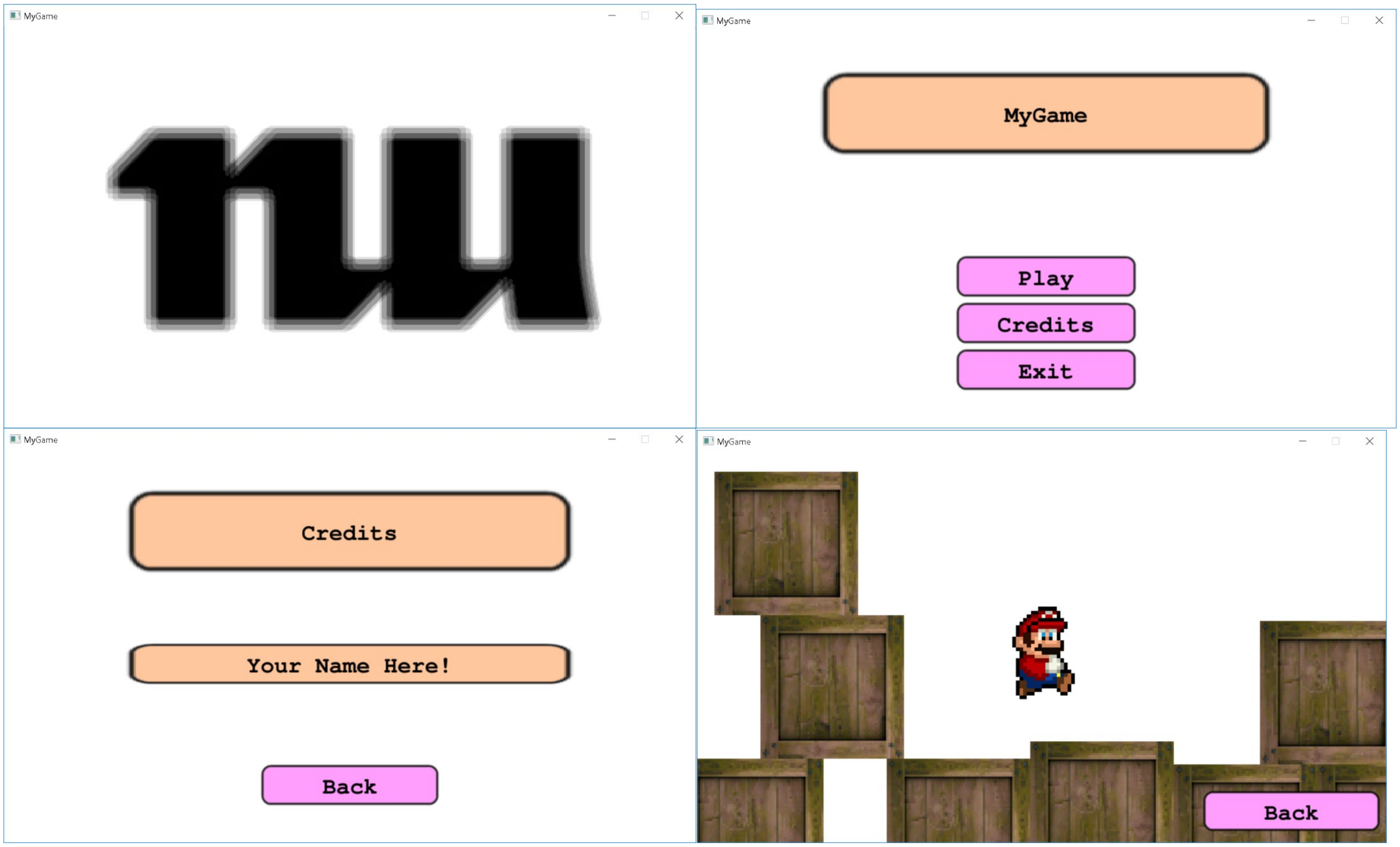
First, set the **Nu** project at the **StartUp** project, then run it by pressing the **|> Start** button in Visual Studio.

When the program runs, it will ask you if you would like to make a new game project and what you would like the name of the project to be. Once you answer these questions, it will create the desired project in the **./Projects** folder in a sub-folder of the given project name and add the new project to **Nu.sln**.

If you're working in Visual Studio, you will be prompted to reload the solution, which you should do. Now you can build and run the new project by setting it as the **StartUp** project and then pressing the **|> Run** button.

For those having difficulty creating a new project (such as on non-Windows platforms), you might instead just want to use the **UserProject** for your game. You can alter its code however you like, but do be aware this is a workaround and you may have to deal with merges as it is kept up to date. Windows users, try to use the previous project creation method instead of **UserProject**.

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
module Program =

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

        // 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 }

        // use the default world config with the above SDL config
        let worldConfig = { WorldConfig.defaultConfig with SdlConfig = sdlConfig }

        // initialize Nu
        Nu.init worldConfig.NuConfig

        // run the engine with the given config and plugin
        World.run worldConfig (MyPlugin ())
```

All this code initializes Nu and instantiates the game engine.

Next is **MyGamePlugin.fs** -

```
namespace MyGame
open System
open Prime
open Nu
open MyGame

// this is a plugin for the Nu game engine that directs the execution of your application and editor
type MyGamePlugin () =
    inherit NuPlugin ()

    // this exposes different editing modes in the editor
    override this.EditModes =
        Map.ofSeq
            [("Title", fun world -> Simulants.Game.SetModel Title world)
             ("Credits", fun world -> Simulants.Game.SetModel Credits world)
             ("Gameplay", fun world -> Simulants.Game.SetModel (Gameplay Playing) world)]
```

NuPlugin has many functions that can be overridden, but the **EditModes** function allow the Nu game to be configured by the editor Gaia to run the game in one of many user-defined states. The first mode, **"Title"**, runs the game with the Title screen showing by default, **"Credits"** with the Credits screen showing, and **"Gameplay"** with the Gameplay screen showing. You can add new modes for each part of your game you want to edit / debug.

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

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

[<AutoOpen>]
module MyGame =

    // this is our Elm-style model type. It determines what state the game is in. To learn about the
    // Elm-style in Nu, see here - https://vsyncronicity.com/2020/03/01/a-game-engine-in-the-elm-style/
    type Model =
        | Splash
        | Title
        | Credits
        | Gameplay of Gameplay

    // this is our Elm-style message type.
    type Message =
        | ShowTitle
        | ShowCredits
        | ShowGameplay
        | Update
        interface Nu.Message

    // this is our Elm-style command type. Commands are used instead of messages when explicitly
    // updating the world is involved.
    type Command =
        | Exit
        interface Nu.Command

    // this extends the Game API to expose the above model.
    type Game with
        member this.GetModel world = this.GetModelGeneric<Model> world
        member this.SetModel value world = this.SetModelGeneric<Model> value world
        member this.Model = this.ModelGeneric<Model> ()
```

```
// this is the game dispatcher that is customized for our game. In here, we create screens as
// content and bind them up with events and properties.
type MyGameDispatcher () =
    inherit GameDispatcher<Model, Message, Command> (Splash)

    // here we define the game's properties and event handling
    override this.Initialize (model, _) =
        [Game.DesiredScreen :=
            match model with
            | Splash -> Desire Simulants.Splash
            | Title -> Desire Simulants.Title
            | Credits -> Desire Simulants.Credits
            | Gameplay gameplay ->
                match gameplay with
                | Playing -> Desire Simulants.Gameplay
                | Quitting | Quit -> Desire Simulants.Title
         match model with Gameplay gameplay -> Simulants.Gameplay.Gameplay := gameplay | _ -> ()
         Game.UpdateEvent => Update
         Simulants.Splash.DeselectingEvent => ShowTitle
         Simulants.TitleGuiCredits.ClickEvent => ShowCredits
         Simulants.TitleGuiPlay.ClickEvent => ShowGameplay
         Simulants.TitleGuiExit.ClickEvent => Exit
         Simulants.CreditsGuiBack.ClickEvent => ShowTitle]

    // here we handle the above messages
    override this.Message (model, message, _, world) =
        match message with
        | ShowTitle -> just Title
        | ShowCredits -> just Credits
        | ShowGameplay -> just (Gameplay Playing)
        | Update ->
            match model with
            | Gameplay gameplay ->
                let gameplay' = Simulants.Gameplay.GetGameplay world
                if gameplay =/= gameplay' then just (Gameplay gameplay') else just model
            | _ -> just model

    // here we handle the above commands
    override this.Command (_, command, _, world) =
        match command with
        | Exit -> just (World.exit world)

    // here we describe the content of the game, including all of its screens.
    override this.Content (_, _) =
        [Content.screen Simulants.Splash.Name (Slide (Constants.Dissolve.Default, Constants.Slide.Default, None, Simulants.Title)) [] []
         Content.screenWithGroupFromFile Simulants.Title.Name (Dissolve (Constants.Dissolve.Default, None)) "Assets/Gui/Title.nugroup" [] []
         Content.screenWithGroupFromFile Simulants.Credits.Name (Dissolve (Constants.Dissolve.Default, None)) "Assets/Gui/Credits.nugroup" [] []
         Content.screen<GameplayDispatcher> Simulants.Gameplay.Name (Dissolve (Constants.Dissolve.Default, None)) [] []]
```

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

https://vsyncronicity.com/2020/03/01/a-game-engine-in-the-elm-style/

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 **Gameplay.fs** -

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

[<AutoOpen>]
module Gameplay =

    // this is our Elm-style model type representing gameplay.
    type Gameplay =
        | Playing
        | Quitting
        | Quit

    // this is our Elm-style message type.
    type GameplayMessage =
        | StartQutting
        | FinishQuitting
        interface Message

    // this is our Elm-style command type. Commands are used instead of messages when things like physics are
involved.
    type GameplayCommand =
        | Update
        | PostUpdateEye
        | Jump
        | Nop
        interface Command

    // this extends the Screen API to expose the above Gameplay model.
    type Screen with
        member this.GetGameplay world = this.GetModelGeneric<Gameplay> world
        member this.SetGameplay value world = this.SetModelGeneric<Gameplay> value world
        member this.Gameplay = this.ModelGeneric<Gameplay> ()

    // this is the screen dispatcher that defines the screen where gameplay takes place
    type GameplayDispatcher () =
        inherit ScreenDispatcher<Gameplay, GameplayMessage, GameplayCommand> (Quit)

        // here we define the screen's properties and event handling
        override this.Initialize (_, _) =
            [Screen.UpdateEvent => Update
             Screen.PostUpdateEvent => PostUpdateEye
             Screen.DeselectingEvent => FinishQuitting
             Game.KeyboardKeyDownEvent =|> fun evt -> if evt.Data.KeyboardKey = KeyboardKey.Up && not
evt.Data.Repeated then Jump else Nop]

        // here we handle the above messages
        override this.Message (_, message, _, _) =
            match message with
            | StartQutting -> just Quitting
            | FinishQuitting -> just Quit
```

```fsharp
// here we handle the above commands
override this.Command (_, command, _, world) =
    match command with
    | Update ->
        let physicsId = Simulants.GameplayScenePlayer.GetPhysicsId world
        let world =
            if World.isKeyboardKeyDown KeyboardKey.Left world then
                if World.isBodyOnGround physicsId world
                then World.applyBodyForce (v3 -2500.0f 0.0f 0.0f) physicsId world
                else World.applyBodyForce (v3 -750.0f 0.0f 0.0f) physicsId world
            elif World.isKeyboardKeyDown KeyboardKey.Right world then
                if World.isBodyOnGround physicsId world
                then World.applyBodyForce (v3 2500.0f 0.0f 0.0f) physicsId world
                else World.applyBodyForce (v3 750.0f 0.0f 0.0f) physicsId world
            else world
        just world
    | Jump ->
        let physicsId = Simulants.GameplayScenePlayer.GetPhysicsId world
        if World.isBodyOnGround physicsId world then
            let world = World.playSound Constants.Audio.SoundVolumeDefault (asset "Gameplay" "Jump") world
            let world = World.applyBodyForce (v3 0.0f 140000.0f 0.0f) physicsId world
            just world
        else just world
    | PostUpdateEye ->
        if World.getAdvancing world then
            let characterCenter = Simulants.GameplayScenePlayer.GetCenter world
            let world = World.setEyePosition2d characterCenter.V2 world
            just world
        else just world
    | Nop -> just world

// here we describe the content of the game including the level, the hud, and the player
override this.Content (gameplay, _) =

    [// the gui group
     Content.group Simulants.GameplayGui.Name []
         [Content.button Simulants.GameplayGuiQuit.Name
             [Entity.Text == "Quit"
              Entity.Position == v3 260.0f -260.0f 0.0f
              Entity.Elevation == 10.0f
              Entity.ClickEvent => StartQutting]]

     // the player and scene groups while playing
     match gameplay with
     | Playing | Quitting ->
        Content.groupFromFile Simulants.GameplayScene.Name "Assets/Gameplay/Scene.nugroup" []
            [Content.sideViewCharacter Simulants.GameplayScenePlayer.Name
                [Entity.Position == v3 0.0f 0.0f 0.0f
                 Entity.Size == v3 108.0f 108.0f 0.0f]]
     | Quit -> ()]
```

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

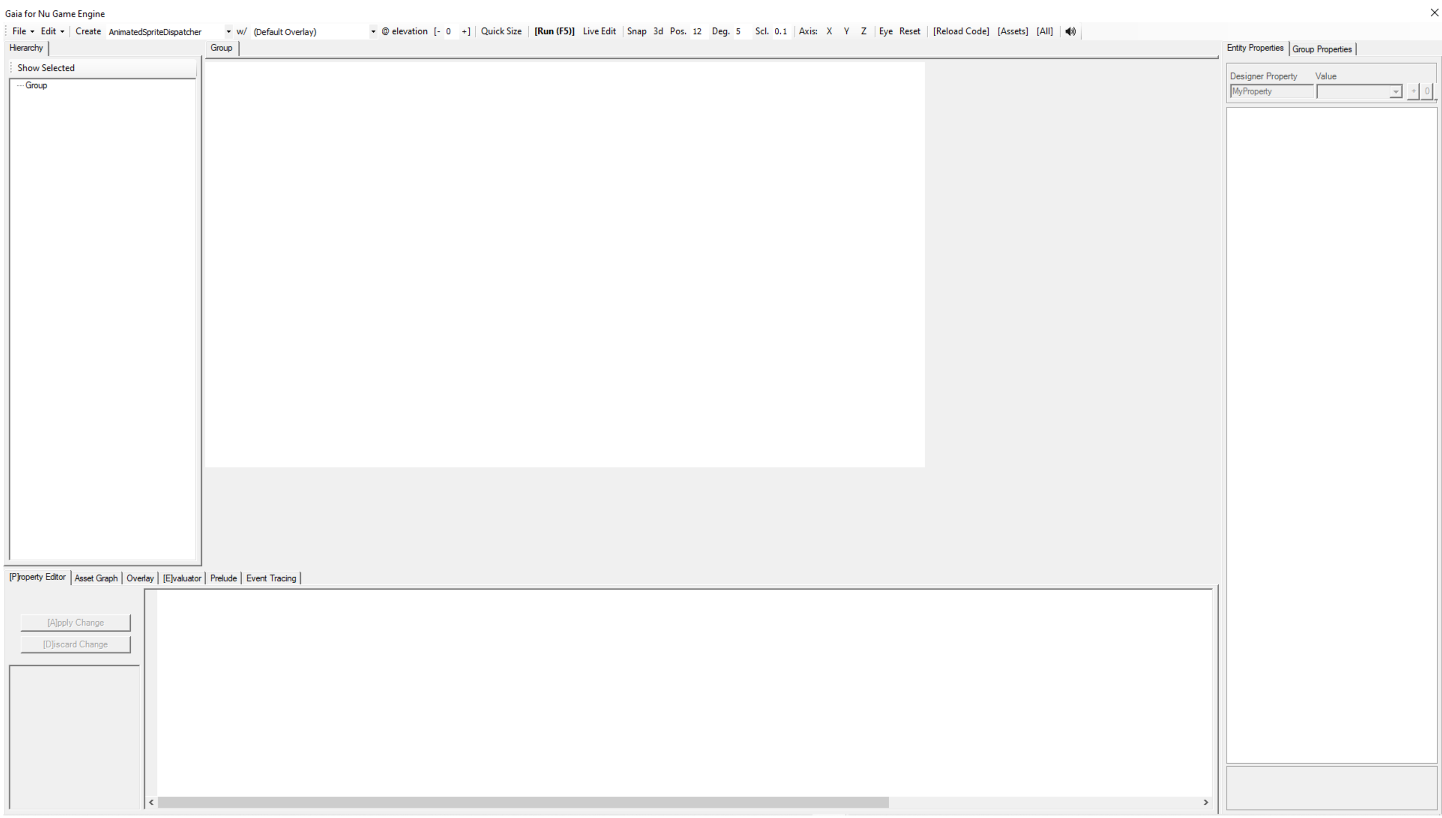
```
namespace MyGame
open System
open Prime
open Nu

// this module provides global handles to the game's key simulants.
// having a Simulants module for your game is optional, but can be nice to avoid duplicating string literals across
// the code base.
[<RequireQualifiedAccess>]
module Simulants =

    // splash screen
    let Splash = Screen "Splash"

    // title screen
    let Title = Screen "Title"
    let TitleGui = Title / "Gui"
    let TitleGuiPlay = TitleGui / "Play"
    let TitleGuiCredits = TitleGui / "Credits"
    let TitleGuiExit = TitleGui / "Exit"

    // credits screen
    let Credits = Screen "Credits"
    let CreditsGui = Credits / "Gui"
    let CreditsGuiBack = CreditsGui / "Back"

    // gameplay screen
    let Gameplay = Screen "Gameplay"
    let GameplayGui = Gameplay / "Gui"
    let GameplayGuiQuit = GameplayGui / "Quit"
    let GameplayScene = Gameplay / "Scene"
    let GameplayScenePlayer = Gameplay / "Player"
```

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 –

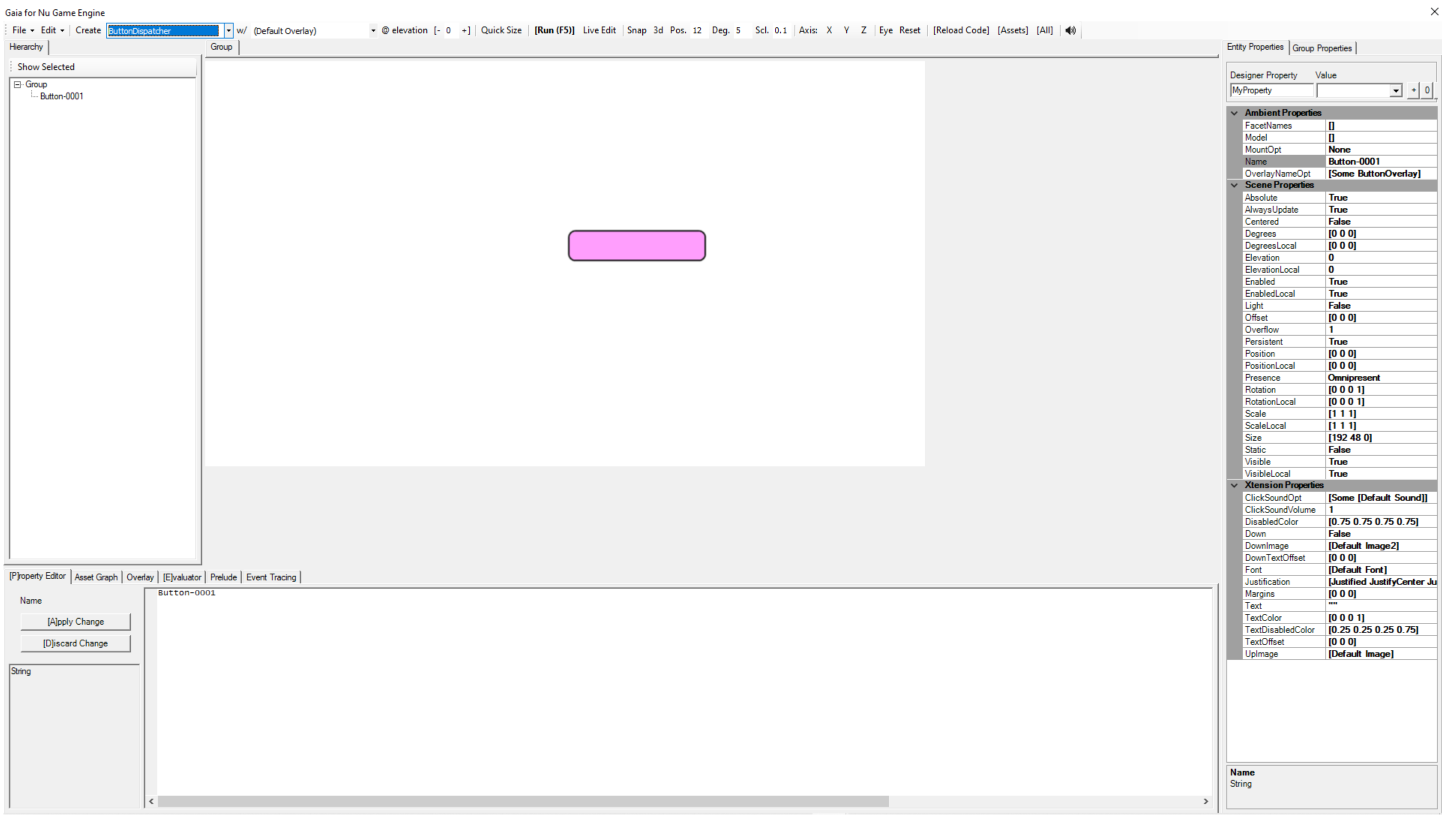


Run Gaia by setting the **Gaia project** as the **StartUp Project** in Visual Studio, and then running.

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).

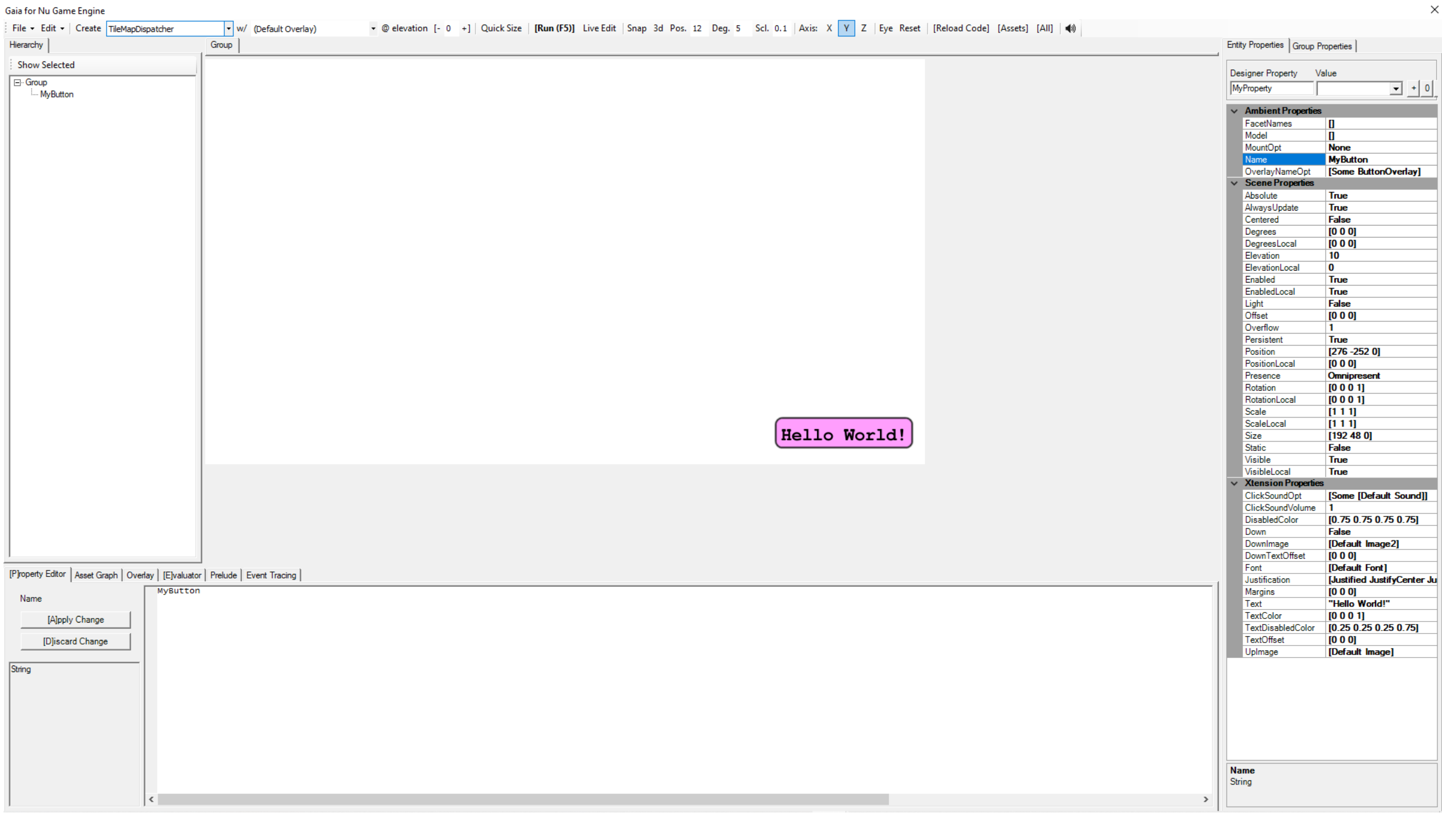
Here we will just cancel the dialog and play with the dispatchers / facets that come out-of-the-box.

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 -



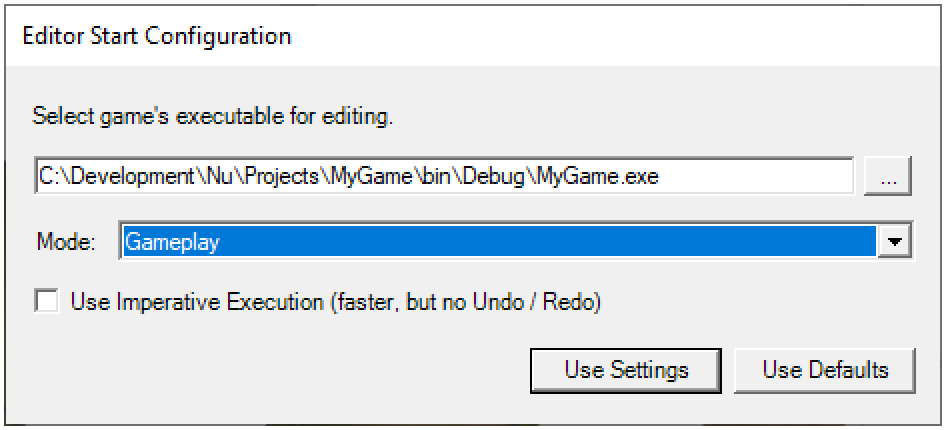
Let's now try putting Gaia in **Run (F5)** mode so that we can test that our button clicks as we expect. Toggle on the **Run (F5)** button at the top, then click on the button you just created. You can also move the button while the game is running by toggling on **Live Edit** mode.

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

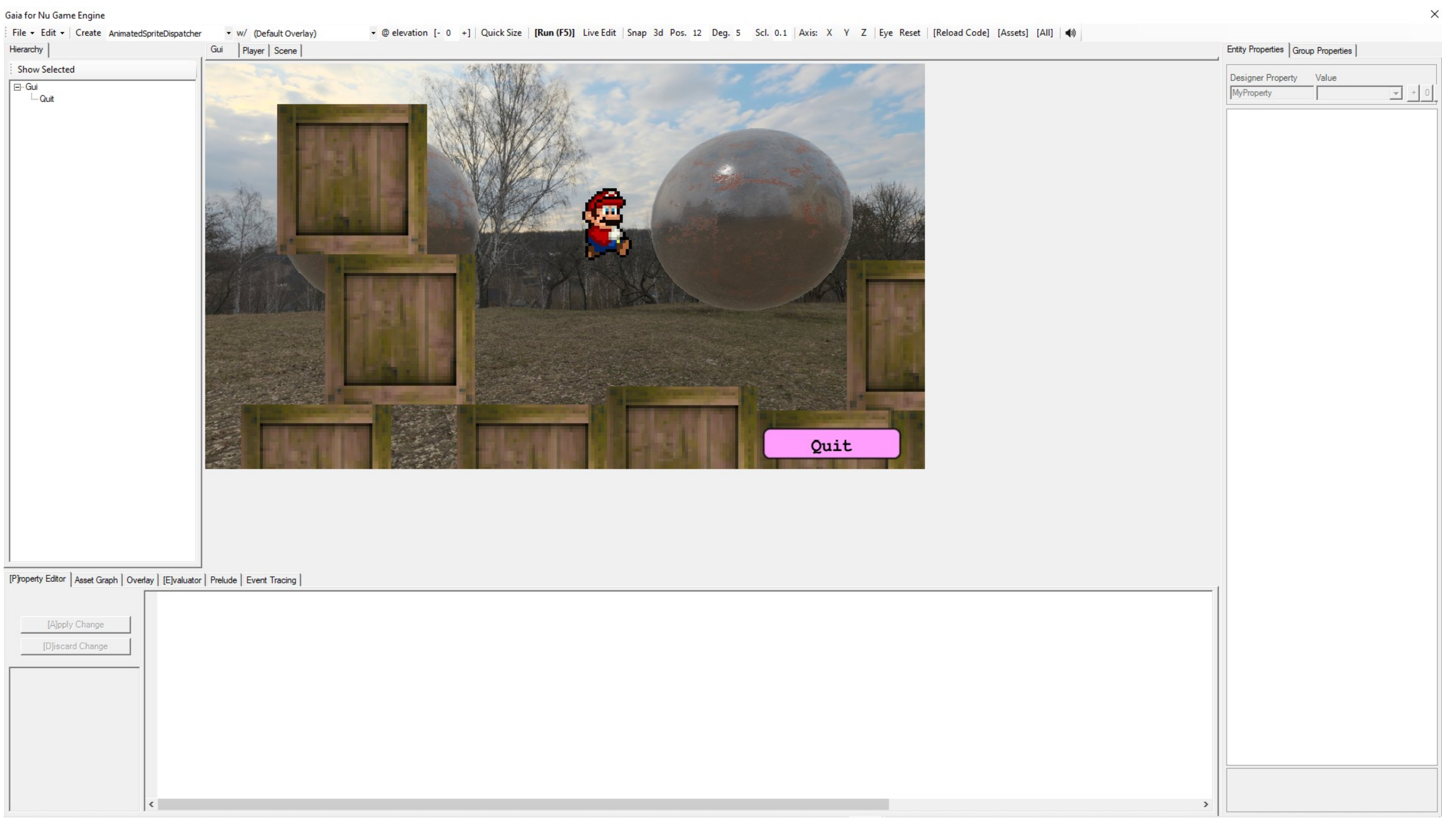
Loading Your Game in Gaia

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, even undoing and redoing gameplay as you play. 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.

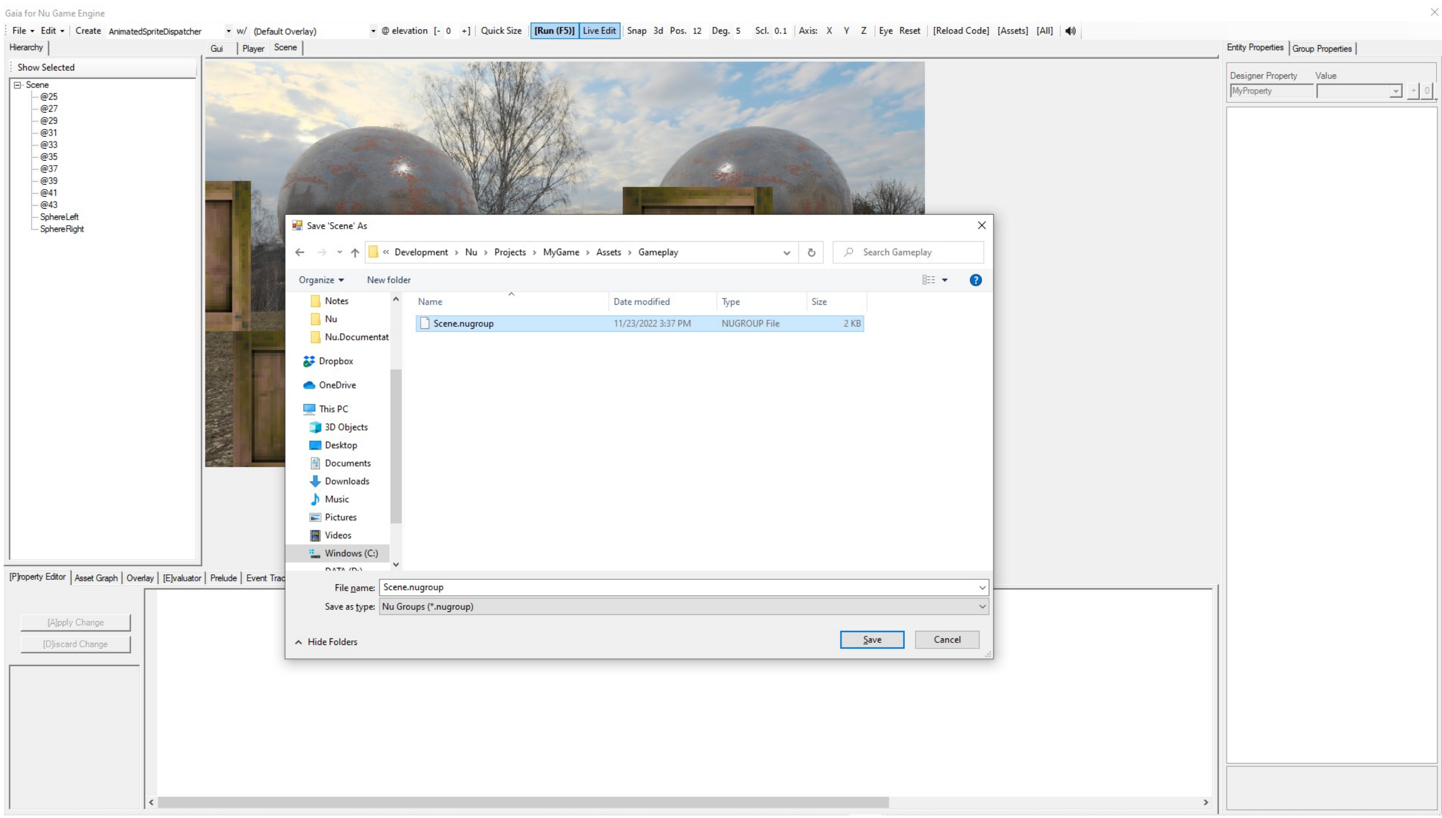
Once we're ready start a new session of Gaia and fill out the start dialog like so -



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



By selecting the tab of a containing Group at the top left, you can edit that Group'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. To save your changes to the **Scene** group, select its tab, press **Ctrl+S** or use **File→Save Group** and save it over the existing group file like so -



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.

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 box with physics and a Gui button – they are both built 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.

Group

Groups represent logical collections of entities that can be combined to make up a Screen. Each group has a Visible property that can be used to hide or show all of its entities.

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, Group, 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/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** 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 group 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 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. If you need a child simulant to communicate with a parent simulant tho, you can build new events for the child to publish and the parent to listen to with the Elmish **=>** and **=|>** operators.

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.

Xtensions Under the Hood

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.

Creating a new Simulant Property with Xtensions

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 to expose new simulant properties when you want to store data outside of the simulant's Elmish model type.

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 world : single = this.Get (nameof this.Density) world
    member this.SetDensity (value : single) world = this.Set (nameof this.Density) value world
    member this.Density = lens (nameof this.Density) this this.GetDensity this.SetDensity
```

- 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 Elmish / MVU 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, but are totally stateless. So they're not really objects in the object-oriented sense, but rather a convenient way that Nu's borrows dispatch polymorphism from .NET's object-oriented constructs. Overriding a dispatcher's methods is how we hook our simulant's custom behavior into the engine.

The **Register** method allows you to customize what happens to the simulant (and the world) when it is added to the world. **Unregister** allows you to customize what happens when it is removed. **Update** is your typical update callback, and **PostUpdate** is well, the post-update. **Actualize** is what can be implemented if you have some custom rendering that you want to implement.

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

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

        abstract member Physics : Vector3 * Quaternion * Vector3 * Vector3 * 'model * Entity * World -> Signal list *
'model
        default this.Physics (_, _, _, _, model, _, _) = just model

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

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

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

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

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

Facets

What if we need some form a composition in order to reuse behaviors among different entities?

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

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 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 (nameof this.StaticImage) world
        member this.SetStaticImage (value : Image AssetTag) world = this.Set (nameof this.StaticImage) value world
        member this.StaticImage = lens (nameof this.StaticImage) this this.GetStaticImage this.SetStaticImage
        member this.GetInsetOpt world : Box2 option = this.Get (nameof this.InsetOpt) world
        member this.SetInsetOpt (value : Box2 option) world = this.Set (nameof this.InsetOpt) value world
        member this.InsetOpt = lens (nameof this.InsetOpt) this this.GetInsetOpt this.SetInsetOpt
        member this.GetColor world : Color = this.Get (nameof this.Color) world
        member this.SetColor (value : Color) world = this.Set (nameof this.Color) value world
        member this.Color = lens (nameof this.Color) this this.GetColor this.SetColor
        member this.GetBlend world : Blend = this.Get (nameof this.Blend) world
        member this.SetBlend (value : Blend) world = this.Set (nameof this.Blend) value world
        member this.Blend = lens (nameof this.Blend) this this.GetBlend this.SetBlend
        member this.GetGlow world : Color = this.Get (nameof this.Glow) world
        member this.SetGlow (value : Color) world = this.Set (nameof this.Glow) value world
        member this.Glow = lens (nameof this.Glow) this this.GetGlow this.SetGlow
        member this.GetFlip world : Flip = this.Get (nameof this.Flip) world
        member this.SetFlip (value : Flip) world = this.Set (nameof this.Flip) value world
        member this.Flip = lens (nameof this.Flip) this this.GetFlip this.SetFlip

    type StaticSpriteFacet () =
        inherit Facet (false)

        static member Properties =
            [define Entity.StaticImage Assets.Default.Image6
             define Entity.Color Color.One
             define Entity.Blend Transparent
             define Entity.Glow Color.Zero
             define Entity.InsetOpt None
             define Entity.Flip FlipNone]

        override this.Render (entity, world) =
            let mutable transform = entity.GetTransform world
            let perimeter = transform.Perimeter
            let staticImage = entity.GetStaticImage world
            World.enqueueRenderLayeredMessage2d
                { Elevation = transform.Elevation
                  Horizon = perimeter.Position.Y
                  AssetTag = AssetTag.generalize staticImage
                  RenderDescriptor2d =
                    SpriteDescriptor
                        { Transform = transform
                          InsetOpt = match entity.GetInsetOpt world with Some inset -> ValueSome inset | None -> ValueNone
                          Image = staticImage
                          Color = entity.GetColor world
                          Blend = entity.GetBlend world
                          Glow = entity.GetGlow world
                          Flip = entity.GetFlip world }}
                world

        override this.GetQuickSize (entity, world) =
            match Metadata.tryGetTextureSizeF (entity.GetStaticImage world) with
            | Some size -> size.V3
            | None -> Constants.Engine.EntitySize2dDefault
```

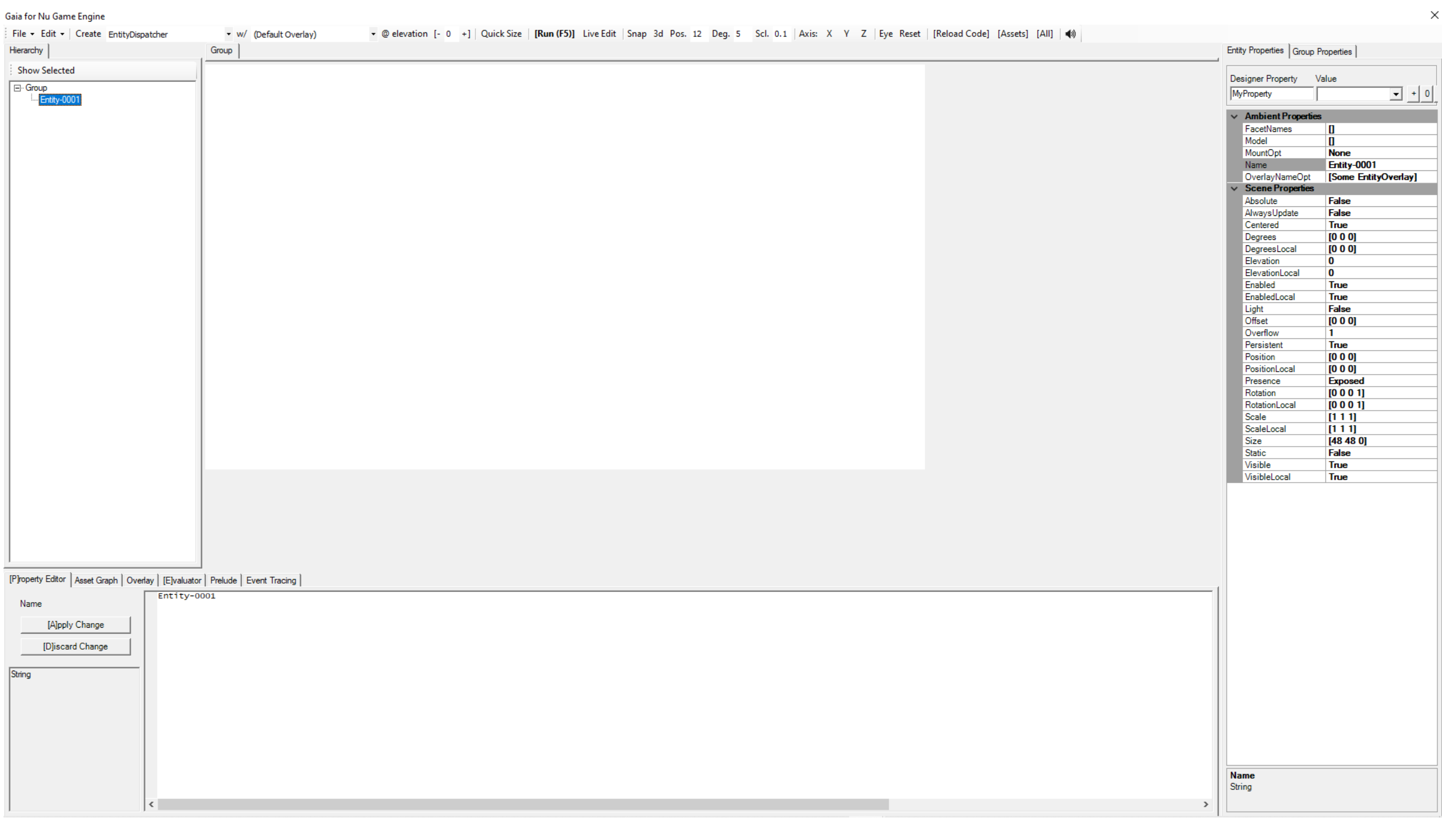
As you may see, the **StaticSpriteFacet** is used to define simple, static sprite rendering behavior for an entity.

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 Facets =
            [typeof<RigidBodyFacet>
             typeof<SpriteFacet>]
```

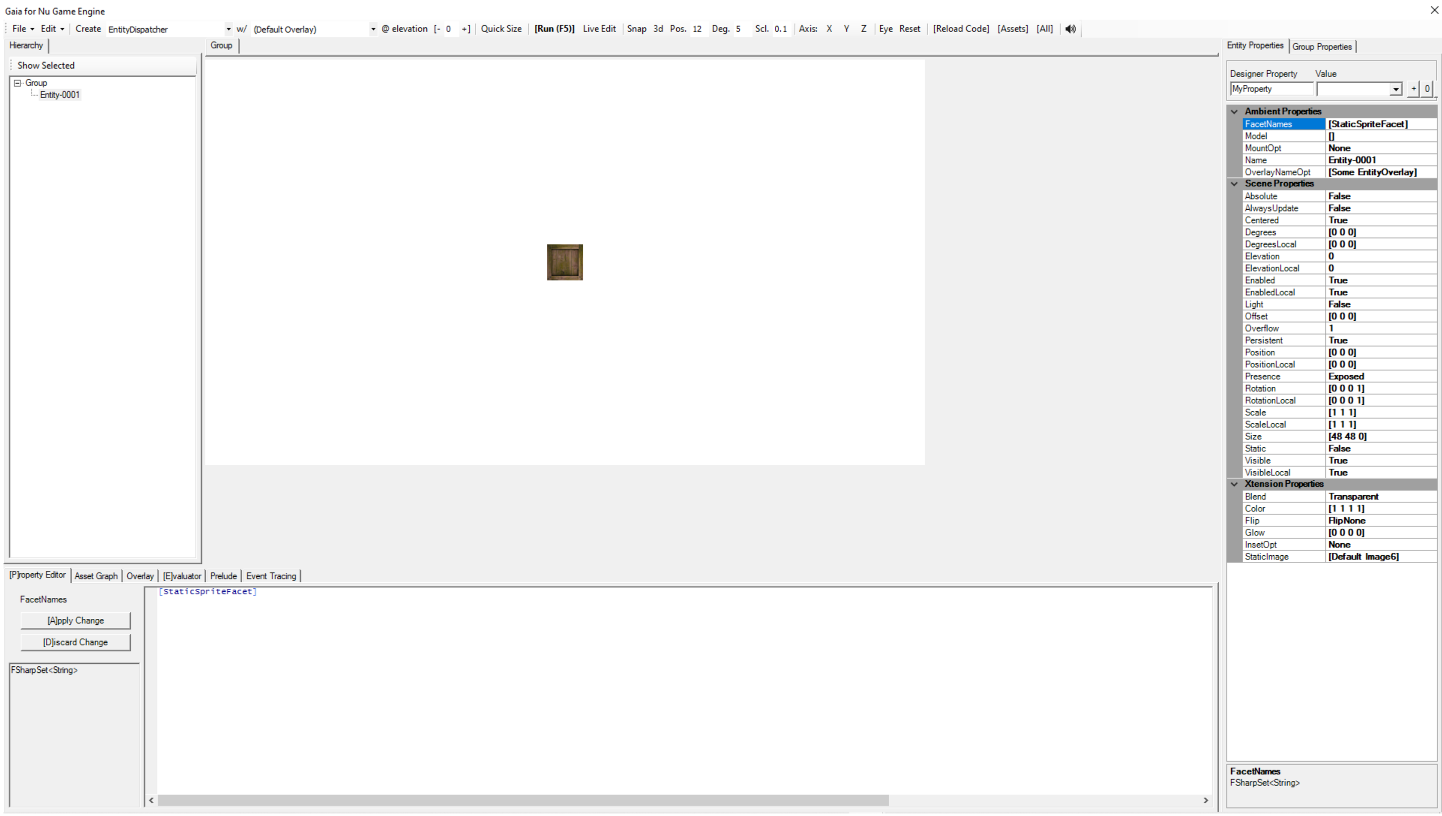
Additionally, facets can be dynamically added to a removed from an entity in Gaia simply by changing the **FacetNames** property. Let's take a look.

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



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 **[StaticSpriteFacet RigidBodyFacet]**, and then toggling on the **Run (F5)** button.

It falls away! Press **Ctrl+Z** to undo gameplay to get it back. 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!

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 new project’s **AssetGraph.nuag** file that ultimately defines a game’s Asset Graph –

```
[[Default
  [[Asset Font "Assets/Default/FreeMonoBold.024.ttf" [Render2d] []]
   [Assets Assets/Default [psd] [Render2d Render3d] [PsdToPng]]
   [Assets Assets/Default [bmp png jpg jpeg tif tiff] [Render2d Render3d] []]
   [Assets Assets/Default [cbm fbx obj mtl glsl] [Render3d] []]
   [Assets Assets/Default [wav ogg] [Audio] []]
   [Assets Assets/Default [nueffect nuscript csv] [Symbol] []]
   [Assets Assets/Default [nugroup tsx tmx] [] []]]]
 [Gui
  [[Assets Assets/Gui [psd] [Render2d Render3d] [PsdToPng]]
   [Assets Assets/Gui [bmp png jpg jpeg tif tiff] [Render2d Render3d] []]
   [Assets Assets/Gui [cbm fbx obj mtl glsl] [Render3d] []]
   [Assets Assets/Gui [wav ogg] [Audio] []]
   [Assets Assets/Gui [nueffect nuscript csv] [Symbol] []]
   [Assets Assets/Gui [nugroup tsx tmx] [] []]]]
 [Gameplay
  [[Assets Assets/Gameplay [psd] [Render2d Render3d] [PsdToPng]]
   [Assets Assets/Gameplay [bmp png jpg jpeg tif tiff] [Render2d Render3d] []]
   [Assets Assets/Gameplay [cbm fbx obj mtl glsl] [Render3d] []]
   [Assets Assets/Gameplay [wav ogg] [Audio] []]
   [Assets Assets/Gameplay [nueffect nuscript csv] [Symbol] []]
   [Assets Assets/Gameplay [nugroup tsx tmx] [] []]]]]
```

This file uses Nu’s s-expression syntax. Why s-expression syntax? Because - https://vsyncronicity.com/2020/03/01/s-expression-the-ultimate-format/

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 message' to unload them via the corresponding **World.unloadRenderPackage2d/3d** 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, you can define it with the **nonPersistent** function rather than the **define** function in the type's **Properties** list like in the following snippet -

```
static member Properties =
    [define Entity.AnimationDelay 3L
     define Entity.AnimationSheet Assets.Gameplay.PlayerImage
     nonPersistent Entity.LastTimeOnGround Int64.MinValue
     nonPersistent Entity.LastTimeJump Int64.MinValue]
```

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. User-defined overlays, called 'overlay routes', 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 route with a matching name is defined with values set to the type's **Properties**.

Let's look at the OmniBlade's overlay definitions –

```
[[OmniButtonOverlay
  [ButtonOverlay]
  [ButtonDispatcher]
  [[ClickSoundOpt [Some [Gui Affirm]]]
   [DownImage [Gui ButtonDown]]
   [Font [Gui Font]]
   [TextColor [1 1 1 1]]
   [TextDisabledColor [.75 .75 .75 .75]]
   [UpImage [Gui ButtonUp]]]]
 [OmniTextOverlay
  [TextOverlay]
  [TextDispatcher]
  [[Font [Gui Font]]
   [TextColor [1 1 1 1]]
   [TextDisabledColor [.75 .75 .75 .75]]]]
 [OmniToggleOverlay
  [ToggleOverlay]
  [ToggleDispatcher]
  [[ToggleSoundOpt [Some [Gui Affirm]]]
   [Font [Gui Font]]
   [TextColor [1 1 1 1]]
   [TextDisabledColor [.75 .75 .75 .75]]]]]
```

Let's understand the **ButtonOverlay** in isolation -

```
[OmniButtonOverlay
  [ButtonOverlay]
  [ButtonDispatcher]
  [[ClickSoundOpt [Some [Gui Affirm]]]
   [DownImage [Gui ButtonDown]]
```

```
[Font [Gui Font]]
[TextColor [1 1 1 1]]
[TextDisabledColor [.75 .75 .75 .75]]
[UpImage [Gui ButtonUp]]]]
```

Obviously enough, the first field, **OmniButtonOverlay**, is the name of the overlay.

The second field, **`[ButtonOverlay]`**, is the list of overlays that the overlay will inherit properties from. This list is composed of a single 'intrinsic' overlay composed of the default properties of the ButtonDispatcher. It's called an 'intrinsic' overlay because it is automatically created by the engine.

The third field, **`[ButtonDispatcher]`**, is the list of dispatchers that the overlay will be automatically applied to.

The remaining list is the set of properties that will be overlaid and their overlaid values.

Overlays avoid a ton of duplication while allowing changes to them to automatically propagate to the entities to which they are applied.