Simple fruit ninja style game:

Data created in the editor:

1. Define Menu Screens, Intro screens, etc…
2. Define Game Screen:
   1. Create properties on the Screen for “Score” & “Time”. Initialize the values
   2. Create a few gameObject templates for the various fruits, bombs, etc… assigning visuals and collision proxies to them, and assigning properties for “points”.
   3. Create a gameObject template for “slice swoosh”, which has a procedural collision proxy based on its runtime specified size. Create a notification receiver component on the object (essentially registers it for notifications) and attach a GameObjectScript which checks for CollisionNotifications, and handles the collisions. Essentially it removes whatever other object you made contact with from the screen and adds the ‘points’ value to the global ‘score’ property. Possibly playing an explosion animation on the way out.
   4. Define a user action initiated by “mouse/touch down” that generates a “Swipe Start” Action Notification with the position.
   5. Define a user action initiated by “mouse/touch move” that generates a “Swiping” Action Notification with the movement.
   6. Define a user action initiated by “mouse/touch up” that generates a “Swipe End” Action Notification with the new position.
   7. Create an invisible gameObject for the player, register it for UserActionNotifications and bind a GameObjectScript that processes and tracks the swipe user actions. On Swipe End, it uses the GameHost to CreateObject on the “slice swoosh” template with the swiping data
   8. Define a ScreenUpdateScript which tracks the global “time” property and once it reaches 0, ends the game. Your final score is stored in the global “Score” property. It can then call ChangeScreen to take you to the next screen.

Bake the data in the editor, then run the game and enjoy… Notice no code in the .exe needed to be written. All data/scripts. Now, we might support C++ as a scripting language, in which case we’d compile all the scripts into a user .dll and hook some entry points, but the core game .exe still wouldn’t change.

Generic single player FPS

Data created in editor:

1. Define Start up screen & sequence
   1. Model a UI Layout panel which has a video control in it
   2. Assign some number of videos to the control, such as ‘studio intro’, and ‘Lucid is awesome’ (this has the effect of preloading and/or streaming both into memory so there’s no hiccup between). Link the completed event of the video control to switch to the second video. The next completed event (of the second movie) initiates a ChangeScreen to the main menu.
   3. Register a UserAction which maps some set of buttons to a “Continue” user action notification
   4. Attach a ScreenUpdateScript which calls ScreenHost.BeginLoadingScreen(“main menu”) on first run, and then tracks that. Once the screen is loaded, it accepts a “continue” user action to skip the rest of the movies by initiating a ChangeScreen to the main menu screen.
2. Define Main Menu screen
   1. Model a UI layout panel (think XAML-like experience of marking up UI here)
   2. Put in a menu control, and insert menu items for your various options like “Continue” (marked as initially hidden), “New Game”, “Options”, “Credits”, and “Exit”. Map each of these to a script action.
   3. Note that these controls are highly skinnable, so this doesn’t have to look like a single boxy cheesy menu. We could have the various menu options displayed on sides of a cube for instance, and you would hook the change menu item event in script and rotate the cube to bring the next menu item into view.
   4. Using some simple scripts (or likely a single script with multiple functions) we provide the customized actions for each menu item.
   5. On startup, in the main update for the UI, we check to see if the user has saved data by consulting the Storage object. If so, we light up the continue option (and possibly rearrange)
   6. NewGame, for instance, will be associated with an action which will likely call ChangeScreen to a LoadingScreen, supplying the name of the next level we’d like to load (level 1 in this case). LoadingScreen will then show some animation while it async loads the asset, then transition to it when it’s done.
3. Define Primary Gameplay screen
   1. Create object templates for the various objects in the game. Example:
      1. Gun
         1. Visual attached
         2. Properties “Ammo Type” and “Ammo Count”
         3. Sound Emitter with a collection of Audio Clips
         4. Notification Receiver.
         5. Process “Fire” notifications to trigger the gun firing. Which, depending on the script action & ammo type, can either raycast, spawn bullets with initial muzzle velocities, or spawn a flame thrower particle system out of the front of the gun.
      2. Soldier 1
         1. Solider visual attached
         2. Weapon object created and attached
         3. Various animation data created
         4. Animation player component attached to him
         5. Dictionary of valid animations set on the player.
         6. Sound Emitter attached
         7. Silent Sound Listener attached (doesn’t play sound on speakers, but will be notified when it can hear a sound, for AI purposes)
         8. Collision Proxy attached
         9. NotificationReceiver attached
         10. Create some properties like “Health”, “current weapon”, and “Loot” on him.
         11. Create a GameObjectScript, which might look something like:
             1. OnNotification: If collision notification, and the ‘other’ is a bullet. Read the damage from the bullet object and reduce it from my ‘health’. If sound notification, and originator is player, and I’m in ‘idle/sleep’ AI state, move me to ‘awake/pursue’ state.
             2. AI state: If I’m in pursue, call ScreenHost.FindObject to locate the player. Call ScreenHost.IsLineOfSight() to see if I can see him, and then move in that direction if I know where he is. If I’m in attack state, then fire my gun in the players general direction.
             3. On Change AI State: Check my saved state->animation mapping and change animations if I’m supposed to.
             4. If my health has reached 0, then I’m dead. Queue up gibbing animation (special flag which allows it to continue even when object is removed), and then remove myself from the screen. FindObject() on player and assign him some points (or notify him of his victory and let him decide points). Spawn loot objects according to my loot property in my location.
      3. Player object
         1. Camera component attached
         2. Collision Proxy attached
         3. Register for UserActions. Handle various user actions in the script for this object. For instance, Sending a ‘Fire’ notification to the weapon you’re holding
         4. Etc… mostly similar to soldier above
   2. Define various levels (instances of Gameplay Screen)
      1. Each level has different static mesh data loaded
      2. Each level is comprised of ‘instances’ of the various object templates we defined above, such as enemies, loot, ammo, etc…
      3. We slap a starting & ending region into each level, the ending region has a tag with the name of the next level asset to load after completion.
      4. Some general metadata (properties) attached to each level, like “name”, “episode”, “difficulty”, etc…
      5. Write a ScreenUpdateScript (either per level if needed, or one shared for every level).
         1. Track player’s points across levels.
         2. Register for UserActions. Handle ‘game level’ actions like Pause, Inventory, etc… Example Interaction with Pause:
            1. OnUserAction “Pause”: Make UI layout control visible & give it focus. Possibly animating it in or some other custom visual. Set screen property ‘paused’ so that enemies stop moving and bullets stop flying. But background animations might/continue.

Example of a detailed creation of a single character in a game:

1. In addition to the built in properties Name and Transform, we can create several top level properties such as “Health”, “CurrentWeapon”, “CurrentArmor”, etc…
   1. Properties can be built as either primitive types:
      1. bool, int, float, string, Vector2, Vector3
   2. Or, they can be built as property bags of (name, primitive type) pairs.
   3. They can also be references to another object
2. Add a visual component
   1. Set the visual type to SkinnedMesh. This registers a Skeleton property on the parent object, which can be targeted by animations.
   2. Load a rigged model and associate it with the component. This will run some validation that the model contains skeletal data and make that data available.
   3. If the model has existing material data associated with it, it will be loaded automatically and can be modified. If it has raw texture data, a simple material will be generated for it using the textures. If no texture information exists, a default material is set. You can fully customize the materials.
   4. Configure whether or not you want IK fixups for the mesh. This requires providing bounding volumes around the feet of the skeleton.
3. Add an AnimationPlayer component on the object.
   1. If the model file contained animations, a new animation dictionary is created on the component and the existing animations are loaded in. New animations can be added at any time, and existing ones edited or removed. This is also fully customizable.
   2. It is important to note that the animations target the Skeleton property on the object. They are not directly dependent on the Visual Component.
   3. There are several options on the player. For example, you can set the default blending behavior for new animations (whether they get blended onto the existing animation for a smooth transition, or abruptly change). Each change call has an option to override the default behavior.
   4. There are also options on each animation, such as whether they repeat by default, etc…
   5. The animation player registers for animation related notifications, such as requests by the object to change animations.
4. Add a collision component
   1. A collision proxy defines the collision bounding volume of the object, and some flags which can help filter what objects the proxy can collide with, and what kind of notifications are generated. For instance you can set an object up as a sensor, which doesn’t count as a collision (so other objects won’t get notified), but the sensor gets notified.
   2. Note that having a collision proxy does not necessarily mean the object participates in physics.
   3. Possible shapes include sphere, oriented box, capsule, and triangle mesh. Probably more later.
5. Add a physics component
   1. A physics component provides some core physical properties such as mass and restitution, and will register you into the physics simulation.
6. Add a state machine component
   1. Fill in the state machine component with the specific states that the object supports (all user specified). Each state gets a user provided script function associated with it. That script function is run once per frame during AI/update, and has the ability to cause a state transition to occur.
7. Add an update script component
   1. This is essentially the general purpose AI unit. The use of the state machine to manage state is optional, and you could write your entire AI logic here. However, if you do use the state machine component to organize your logic, note that this script function still runs every time. It’s a great place to put in common logic such as “when my life reaches 0, I’m dead”.
   2. You may optionally register for various notifications here, such as collision notifications, sensor enter/exit notifications, etc…
   3. It is guaranteed to run before state script actions.
8. Add an audio emitter
   1. The audio emitter works very much like the animation player. You build a dictionary of audio clips, or sound effects, and you can configure some basic properties such as volume, range of the emitter, etc...
   2. The audio emitter also registers a private notification sink to listen for requests to play a sound effect.

For the main player, we can add a few other special components, though not all of these components are limited to just the player:

1. Camera component
   1. This allows specifying field of view, perspective or orthographic, and some other useful properties.
   2. The camera has a name/ID and is registered in the screen’s camera directory so that you can toggle between them from any of the scripts anywhere on the screen.
2. In the user’s update script component:
   1. We additionally register for UserActionNotifications. These are notifications raised from user actions, which we define per screen. For instance, we may have mapped the trigger controller button to raise a “FireInput” analog (floating point) notification.
   2. When we define user actions at the screen level, we can have them generate/map certain types of data:
      1. Screen space position (Vector2)
      2. Raycast position (Vector3)
      3. Direction (Vector2)
      4. Analog signal (float)
      5. Digital signal (bool)

M0 – “assets”

1. Create game object templates:
   1. Block – static mesh visual, default material
   2. Dude – static mesh visual, default material
   3. Camera man – no visual, camera component
2. Main gameplay GameScreen
   1. Place an instance of the Block object, with a large scale in the X and Z directions. This acts as our ‘ground’
   2. Place an instance of the dude on top of the block (no physics yet in M0), have this instance sink user action notifications with a script action
   3. Create user action table for the screen
   4. Place an instance of the camera man in the scene. Have this instance sink user action notifications