Squirrel Engine

Base Principles

Squirrel Engine is a 3D game engine featuring dynamic deserialization and physics integration. It uses a scene graph to organise objects in the scene.

Documentation for each class is available here: https://corcso.github.io/SquirrelEngine (Heesh, 2024)

Squirrel Engine is based off of the core idea of having 2 types of objects. Nuts (Nodes, Objects) and Resources

Nuts

Uniquely owned by their parent in the scene graph.
Created programmatically or instantiated from a file.
Used for any world objects such as: Lights, cameras, meshes.

Resources

Shared ownership across users.
Loaded from a file.
Used for resources such as: Meshes, materials, collision shapes.

Platform Support

Currently Squirrel engine supports windows builds with DirectX11 (Microsoft, 2009) for rendering and Jolt (Rouwe, no date) for physics simulation. Although these services have been sufficiently abstracted so that another library could be used instead.

Example Game

When creating a new project the example game project should be used. This has the entire engine with the correct compilation settings as well as a small game example (which can be easily removed) to help you get started and provide code examples.

Nuts

Developers are able to create their own nuts by taking an existing nut as a base class. Developers can then add functionality and custom deserialization to their nut. For example: A player nut, inherited from PhysicsNut. This contains movement functionality and a health taken in from deserialization.

There are 6 nut types which come with the engine. Nut, WorldNut, MeshNut, LightNut, CameraNut and PhysicsNut. Each have their own functionality.

Nut - A nut with only a name and child / parent information. This nut has no transform in 3D space. It has functionality required across all nuts and is the final parent class for all nuts. This nut can be used for things like folders in the scene where objects need to be placed.

WorldNut - A Nut which has a 3D transformation, a location, rotation and scale in the 3D world. All children in the scene graph inherit this transform.

MeshNut - A WorldNut which has a mesh (3D model) and material applied. This is the only renderable nut type.

CameraNut - A WorldNut which is used for cameras. Has a FOV, and a isActive property. Only one camera should be active at a time.

LightNut - A WorldNut which represents a light in the scene. There are 3 lights, Directional, Point and Spot. And the game supports up to 8 lights.

PhysicsNut - A WorldNut which undergoes physics simulation. It must have a CollisionShape attached.

To create your own type of nut for use in the scene you should:

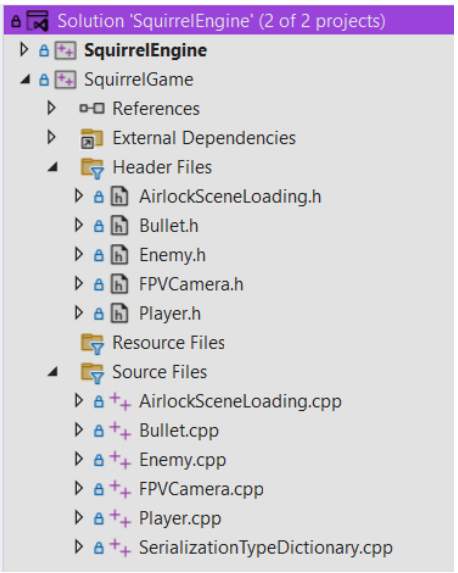
1. Copy the `TEMPLATE.h` and `TEMPLATE.cpp`
2. Replace `CLASSNAME` with your class name of choice
3. Replace `PARENTNAME` with a your chosen parent nut
4. Add the the deserialize function to the SerializationTypeDictionary (see below)

Resources

There are 4 types of resources in Squirrel Engine: meshes, materials, collision shapes and shelled nuts.
Mesh - A 3D model object used for rendering with a MeshNut. Currently only supports 1 object model files. `.obj` is recommended but other file types like .fbx should be supported.
Material - `.mat` - A material which is applied to a mesh, contains attributes like colour and smoothness.
Collision Shape - `.shape` - A shape used by the physics engine to calculate collisions. There are 3 available types, Cube/Cuboid, Sphere and Capsule.
Shelled Nut - `.nut` - A storage resource for holding a serialized nut scene. Used to instantiate nut scenes at runtime.

Development Environment & Getting Started

Squirrel Engine currently uses Microsoft Visual Studio as its sole development environment. You should use the example project as a starting point for your game. This contains a visual studio solution with the correct setup and a small shooter game to serve as an example to do various things.



The solution contains 2 projects. SquirrelEngine and SquirrelGame. SquirrelGame is where you will perform most of your programming.
In the SquirrelGame project folder in file explorer there is also an options.json file and a Resources folder.
The options file contains all the project settings. All options must be set in json format.

JSON Property Name	Usage
Window Name	Name for the window. Appears in the top bar.
Window Size	Pixel size of the rendering window.
On Load Nut	Serialized Nut .nut file to load on start.
Background Color	The background color to be set.
Target FPS	The target FPS for the game. The game is limited to this FPS.

The resources folder is where you will put all your resources and serialized nuts. All resources are referenced by their file path. You are able to create other folders within and outside the resources folder if you wish. Example path:
`./Resources/wood.mat`
All resources with the exception of meshes are JSON files. You are able to use any text editor to edit these files as you wish.

Nut Functions To Override

The following nut functions should be overwritten. They should also call their parent class's respective function first.

- ➔ Ready()
- ➔ Update()
- ➔ LateUpdate()

`Ready()` is called once on a nut when it first enters the main scene tree. If a nut leaves the scene tree and is then re-added, ready will not be called again.
`Update()` is called once per frame
`LateUpdate()` is called once per frame, after all updates.
The static function `Deserialize()` should also be added to ensure your nuts can be loaded from a .nut file. It is best to copy the deserialization function from the template and add any deserialization parameters between the following comments.

```
// YOUR DESERIALIZATION HERE, USE TOWORKON
```

and

```
// Return ownership of the new nut
```

PhysicsNuts also have `OnCollisionStart(PhysicsNut*)` and `OnCollisionEnd(PhysicsNut*)` virtual functions. These should be overridden if you would like to perform certain actions based on collisions. The physics nut pointer passed in is the object the collision has started or ended with. These are called before `Update()`.

Nut Ownership

All nuts are stored within a Pool Allocator. This allows for faster creation and deletion of nuts. Due to nuts varying size, several pool allocators are used, the Pool Allocation Service handles this for you and is able to create unique pointers to pool objects.
Objects in pools must use the UniquePoolPtr unique pointer rather than the STL unique pointer.
All raw pointers to nuts should be considered observers only, and should never be deleted.

When creating a new nut programmatically you should call
`GetPoolAllocationService()->MakeUniquePoolPtr<NUT TYPE>();`

Nuts follow a **strict** ownership. Once they are created their ownership must be passed around with a UniquePoolPtr at all times. When reparenting a nut owned by another nut, you can simply call the `SetParent()` function and pass in the new parent. When reparenting a nut owned by you, you must get an observer pointer (via `.get()`) first and then call `SetParent()` also passing in your UniquePoolPtr, to hand over ownership. This can be seen in figure X.
You can additionally use `AddChild()` to reparent a nut but you **should only call this on nuts you own** as it will not remove the nut as a child from its parent.

.Nut files, Instantiation & Shelled Nuts

For the starting scene of your game you must provide a nut file, everything from here on out can be created programmatically but it is recommended to use a nut file to save time, provide cleaner code and easier editing.
Nut files are JSON files which contain a serialized scene graph. This scene can contain any type of nut, developer or engine defined as well as any depth of children.
Properties available depend on the type of nut being represented but all nuts must have a name and a type.
You are also able to create custom properties which can be deserialized on your own nut types using by editing the Deserialize function.
On the right is a portion of the secondLevel.nut file in the example project. It contains a scene root called “Second Level” which has a static PhysicsNut “Floor” as a child and a LightNut “Spot Light”.
Below is a table containing all the properties which can be set, it is best practice to define all properties even if not using them.

```
{
  "type": "WorldNut",
  "name": "Second Level",
  "position": [0, 0, 40],
  "eulerAngles": [0, 0, 0],
  "scale": [1,1,1],
  "children": [
    {
      "type": "PhysicsNut",
      "name": "Floor",
      "collisionShape": "./Resources/floor.shape",
      "density": 1,
      "elasticity": 0.1,
      "static": true,
      "position": [0, 0, 0],
      "eulerAngles": [0, 0, 0],
      "scale": [1,1,1],
      "children": [{
          "type": "MeshNut",
          "name": "Floor Mesh",
          "mesh": "./Resources/cube.obj",
          "material": "./Resources/floor.mat",
          "position": [0, 0, 0],
          "eulerAngles": [0, 0, 0],
          "scale": [20,1,40],
          "children": []
      }]
    },
    {
      "type": "LightNut",
      "name": "Spot Light",
      "position": [0,5,0],
      "eulerAngles": [-1.6,0,0],
      "scale": [1,1,1],
      "diffuseColor": [0 ,1,0],
```

Property	Type	Usage
`type`	String	The type name used in the SerializationTypeDictionary to create the correct type of nut.
`children`	Array of objects	An array containing all child nuts.

Nut		
name	String	Name for the nut in the scene tree, recommended to be unique.
WorldNut		
position	Array of 3 Numbers (Vector 3)	Local world position of this WorldNut
eulerAngles	Array of 3 Numbers (Vector 3)	Local euler angles of this WorldNut
scale	Array of 3 Numbers (Vector 3)	Local scale of this WorldNut
MeshNut		
mesh	String, Resource File Path	Mesh to use
material	String, Resource File Path	Material to use
LightNut		
diffuseColor	Array of 3 Numbers (Vector 3)	Colour of diffuse light
ambientColor	Array of 3 Numbers (Vector 3)	Colour of ambient light
intensity	Number, decimal	Intensity of diffuse light
ambientIntensity	Number, decimal	Intensity of ambient light
lightType	Enum String, one of: “directional”, “point” or “spot”	The type of light this light nut represents.
spotlightInnerAngle	Angle in radians, number, decimal	The inner cutoff angle for spotlights
spotlightOuterAngle	Angle in radians, number, decimal	The outer cutoff angle for spotlights
linearAttenuation	Number, decimal	Linear attenuation, dims light over linear distance.
quadraticAttenuation	Number, decimal	Quadratic attenuation, dims light over square distance.
CameraNut		
fov	Number, decimal	Field of view of the camera
isActive	Boolean	Boolean to set if the camera is the active renderer or not

PhysicsNut		
static	Boolean	If true the object is considered static and non moving. If false the object is considered dynamic and will move.
elasticity	Number, decimal	Also known as restitution, the number between 0 and 1 which represents bounciness.
density	Number, decimal	Density of the object, determines mass.
collisionShape	String, Resource file path	Path to the .shape file used for this nuts collision shape.

When you load a ShelledNut that only loads the JSON serialized data into memory. The scene itself needs to be instantiated, this creates all the nuts defined in the .nut file and loads all the relevant materials.

To instantiate a shelled nut, call its `Instantiate()` function. This returns a unique pointer to the root nut of the scene giving you ownership. You can then add it to the relevant point in the scene graph if you wish.

Instantiating can be a long process, large instantiations should be handled via the `InstantiateMultithread()` function. This instead returns a promise (via shared pointer), containing a completed boolean and a unique pointer to the root nut of the scene, once it has finished loading.

The developer should poll until the instantiations completion and then take ownership of the scene. An example of this can be seen in the AirlockSceneLoading nut in the example game.

```
// Check the player is inside the airlock area.
Player* player = GetTree()->GetRootNut()->GetNut<Player>("Scene/Main Player");
if (player && promise.get() == nullptr && LenSqrV3(player->GetGlobalPosition() - V3(0, 0, 20)) <= 25) {
    // If so start the scene loading on another thread
    promise = packedScene->InstantiateMultithread();
}

// If we haven't added the scene to the main tree
// And we have a promise
// And that promise shows that the scene has completed loading.
if (!loaded && promise.get() != nullptr && promise->complete) {
    // Set loaded to true
    loaded = true;
    // Get ownership of the new level from the nut.
    UniquePoolPtr<Nut> newLevel = std::move(promise->result);
    // Get an observer to the nut
    Nut* observer = newLevel.get();
    // Hand ownership over to the main tree and place the scene in the main tree.
    observer->SetParent(GetTree()->GetRootNut(), std::move(newLevel));
}
```

Figure X: Snippet from Update() in AirlockSceneLoading

SerializationTypeDictionary

The SerializationTypeDictionary is core to creating a game with Squirrel Engine. It is a map which maps strings to Deserialization functions. The key of the map, the string, should be the type name used in .nut files. The value, the function, should be the corresponding deserialization function.

The SerializationTypeDictionary is defined in the engine but needs to be created in the game. The example game comes with one already.

```
// Squirrel Engine Includes
#include "SquirrelEngine.h"
#include "SerializationTypeDictionary.h"

// ===================================
// PLACE YOUR INCLUDES HERE
#include "FPVCamera.h"
#include "Player.h"
#include "Enemy.h"
#include "Bullet.h"
#include "AirlockSceneLoading.h"
// ===================================

// Dictionary setup
std::unordered_map<std::string, std::function<SQ::UniquePoolPtr<Nut>(Nut*, nlohmann::json)>> SQ::SerializationTypeDictionary({
    // Squirrel Engine Nuts, Editing is not recomended.
    {"Nut", SQ::Nut::Deserialize},
    {"WorldNut", SQ::WorldNut::Deserialize},
    {"MeshNut", SQ::MeshNut::Deserialize},
    {"CameraNut", SQ::CameraNut::Deserialize},
    {"LightNut", SQ::LightNut::Deserialize},
    {"PhysicsNut", SQ::PhysicsNut::Deserialize},

// ===================================
// PLACE YOUR NUT DESERIALIZE FUNCTIONS HERE!
// {"Name Of Type In .nut file", MYCLASS::Deserialize}
    {"FPVCamera", FPVCamera::Deserialize},
    {"Player", Player::Deserialize},
    {"Enemy", Enemy::Deserialize},
    {"Bullet", Bullet::Deserialize},
    {"AirlockSceneLoading", AirlockSceneLoading::Deserialize}
// ===================================

});
```

Deserialize Function

The deserialize function should be a static function which returns a unique nut pointer and takes in a nut pointer and some json data.
If the nut pointer is null, the deserialization function should create a new nut and use that to deserialize on. The deserialization function should call its parent class function before doing any deserialization of parameters.
If the nut pointer is not null, the deserialization function should perform deserialization on that nut pointer instead.

.shape and .mat Resources

Collision shapes and materials are also represented via JSON below are tables which detail their available properties.

.shape		
`type`	Enum, String, One of: “box”, “sphere” or “capsule”	Determines the type of collider shape.
`boxHalfDimensions`	Array of 3 Numbers (Vector 3)	Dimensions from the center to the edge of the box in each direction
`sphereRadius`	Number, decimal	Radius of the sphere.
`capsuleRadius`	Number, decimal	Radius of the capsule hemispheres
`capsuleHalfHeight`	Number, decimal	Length from the center to the edge of a hemisphere for the capsule
.mat		
`diffuse`	Array of 3 Numbers (Vector 3)	Diffuse colour, colour of the object
`specular`	Array of 3 Numbers (Vector 3)	Specular colour, typically white
`specularity`	Number, decimal	Specularity, how sharp the specular highlight is. Must be 1 or above, powers of two recommended. 64 or 128 is a good value.
`smoothness`	Number, decimal	Smoothness, 0 is matte, 1 is smooth

Inherited Transforms & Super local rotations

WorldNuts inherit the transforms of any parent WorldNuts in the scene.
The RotateSuperLocal functions rotate the nut around a local right up or forward direction. The local right, up or forward is the nut's right, up or forward direction rotated only by the local rotation, not any parent rotations. These can be used for things like vertical camera rotation.

Important Information about PhysicsNuts

PhysicsNuts have some properties which are not intuitive, these are explained below.

- ➔ Static or Dynamic must be set before the Nut is placed in the main scene tree. This is handled automatically via deserialization but for manual creation of nuts this should be noted.
- ➔ When changing the attributes of a nut’s collision shape this will change the collision shape for all nuts with said shape file. A duplication of the shape file should be made if the nut will make unique edits to a commonly used collision shape over its lifetime.
- ➔ PhysicsNuts should not be placed as a child of a rotated WorldNut. This is due to a set global rotation and get global rotation function not being currently available. Although PhysicsNuts can successfully inherit position.
- ➔ Applying a scale to a PhysicsNut will not have any effect on its collision shape.