CMP316 - Engine Development - Squirrel Engine

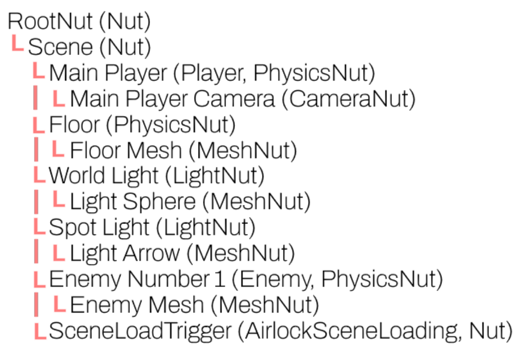
Documentation

Cormac Somerville - 2200592

Base Principles

Squirrel Engine is a 3D game engine featuring dynamic deserialization and physics integration. It uses a scene graph/tree to organise objects in the scene. The game engine is inspired from the design of Godot (Godot Foundation, 2007), featuring a similar object and scene layout.

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



```
RootNut (Nut)
└Scene (Nut)
  └Main Player (Player, PhysicsNut)
  | └Main Player Camera (CameraNut)
  └Floor (PhysicsNut)
  | └Floor Mesh (MeshNut)
  └World Light (LightNut)
  | └Light Sphere (MeshNut)
  └Spot Light (LightNut)
  | └Light Arrow (MeshNut)
  └Enemy Number 1 (Enemy, PhysicsNut)
  | └Enemy Mesh (MeshNut)
  └SceneLoadTrigger (AirlockSceneLoading, Nut)
```

Figure 1: Scene tree with main.nut loaded

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

Nuts	Resources
Uniquely owned by their parent in the scene graph.	Shared ownership across users.
Created programmatically or instantiated from a file.	Loaded from a file.
Used for any world objects such as: Lights, cameras, meshes.	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. 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.
The example features a simple shooter game. WASD to move, Space to jump, Left click to shoot, G displays pool allocator capacity, K turns the player's collider into a sphere.
If the player walks under the red spotlight a second scene will load, this is to simulate an “airlock” loading mechanic used in certain games.

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 could contain movement functionality and a total health, taken in from deserialization.
There are 6 nut types which come with the engine. Nut, WorldNut, MeshNut, LightNut, CameraNut and PhysicsNut. Each has their own functionality.

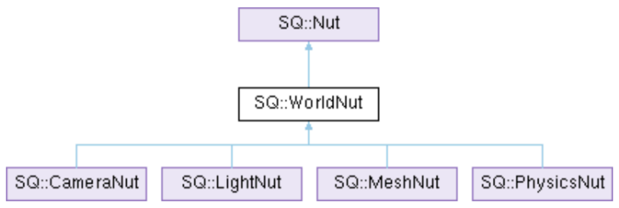


Figure 2: Nut class inheritance diagram. (Heesh, 2024)

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 an isActive property. Only one camera should be active at a time.
LightNut - A WorldNut which represents a light in the scene. There are 3 light types, Directional, Point and Spot. The engine 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` files
2. Replace `CLASSNAME` with your class name of choice
3. Replace `PARENTNAME` with a your chosen parent nut's class name
4. Add the the deserialize function to the SerializationTypeDictionary (see section below)

Resources

There are 4 types of resources in Squirrel Engine: meshes, materials, collision shapes and shelled nuts.

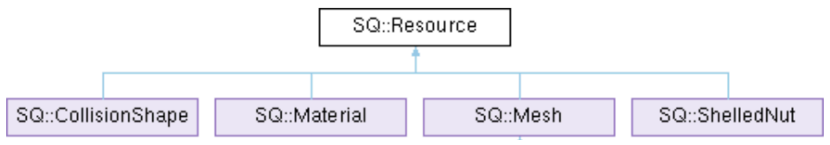
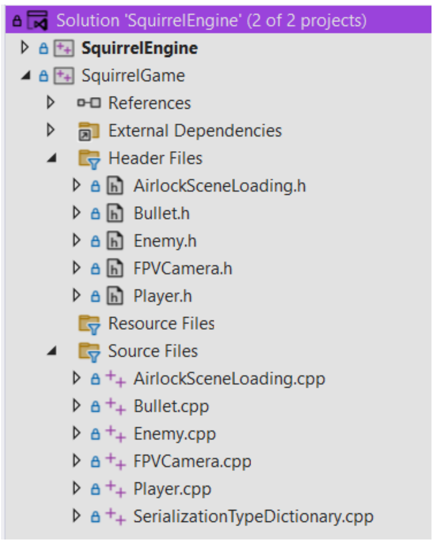


Figure 3: Resource class inheritance diagram. (Heesh, 2024)

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. Assimp is used for loading (Kulling, no date a)
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: Box, Sphere and Capsule.
Shelled Nut - `.nut` - A storage resource for holding a serialized nut scene. Used to instantiate nut scenes at runtime.
Resources should be loaded using the resource manager. Use `GetResourceManager()` from anywhere in your game to retrieve a pointer to the resource manager. Then use `Retrieve()` to retrieve a resource, this will load the resource if required.
Resource references should always be stored with a shared pointer.
Currently resources are not stored in pool allocators, so the STL shared pointer is used.

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.



```
Solution 'SquirrelEngine' (2 of 2 projects)
  SquirrelEngine
  SquirrelGame
    References
    External Dependencies
    Header Files
      AirlockSceneLoading.h
      Bullet.h
      Enemy.h
      FPVCamera.h
      Player.h
    Resource Files
    Source Files
      AirlockSceneLoading.cpp
      Bullet.cpp
      Enemy.cpp
      FPVCamera.cpp
      Player.cpp
      SerializationTypeDictionary.cpp
```

Figure 4: Solution Layout

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.

Figure 5: Options.json file structure

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()`.

Maths

An edited version of handmade math (bvisness, 2016) is used for the maths functionality of Squirrel Engine. It supports two, three and four dimensional vectors as well as square matrices. It also supports quaternion operations. Due to the library supporting C as well as C++ construction of maths objects happens via separate functions, for example `Vec3`'s are constructed with the `V3()` function. `Mat4`'s with `M4()` and `Quat`'s with `Q()`.

Services

There are 7 services part of the engine. These are: Graphics, Physics, Input, Time, PoolAllocationService, ResourceManager and Tree. Graphics & Physics typically are only for engine interaction. Input provides input checking functionality. Time provides functions for delta time and time since start. PoolAllocationService is used to create new nuts, see Nut Ownership below. ResourceManager is used to load resources from a file, stopping them from being loaded twice and unloading them when they are no longer needed. And finally Tree is used for general engine functions like `Quit()`, it also handles the game loop and stores the scene root. Each can be accessed via `Services::GetXXXX()` or the shorthand `GetXXXX()`.

Input

Input is handled via the input service. Use `GetInput()` to access it. Input has functionality for getting key and mouse button states as well as checking mouse position and getting relative mouse movement when the mouse is locked. There are `Key` and `MouseButton` Enums for referencing keys and buttons but characters can also be used for keys.
A Button/Key has 4 states, Up, Down, Pressed and Released. Pressed and Released only trigger on the frame the key is pressed or released.

Nut Ownership

All nuts are stored within a Pool Allocator (Gregory, 2018). 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.
If there isn't a pool with big enough blocks or there is but that pool is full, the Pool Allocation Service will default to using `new`.
UniquePoolPtr supports the use of `new` and `delete` keywords, so one is still returned.
You can edit the pool count, block sizes and block counts in the `PoolAllocationService.h` file.
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()` on that observer, not your unique pointer, also passing in your UniquePoolPtr, to hand over ownership. This can be seen in figure 8.
You can additionally use `AddChild()` to reparent a nut but you **should only call this for adding nuts you own** as it will not remove the nut as a child from its parent.

.Nut files, Instantiation & Shelled Nuts

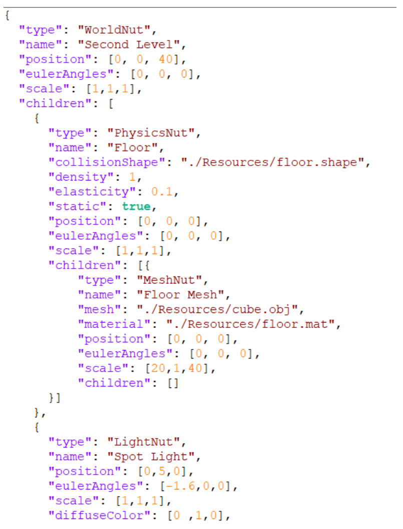
For the starting scene of your game you must provide a .nut file, everything from here on out can be created programmatically but **it is recommended to use a nut file to save time, provide cleaner code and easier editing.**

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.

JSON files are parsed with the nlohmann JSON library (Lohmann, 2013)

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.



```json
{
  "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],
```

Figure 6: Section of secondLevel.nut from the example project

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" as another child.

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**.

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 in radians
`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, a 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. **Required**

Figure 7: Table of engine nut properties

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 resources.

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 the promise 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.

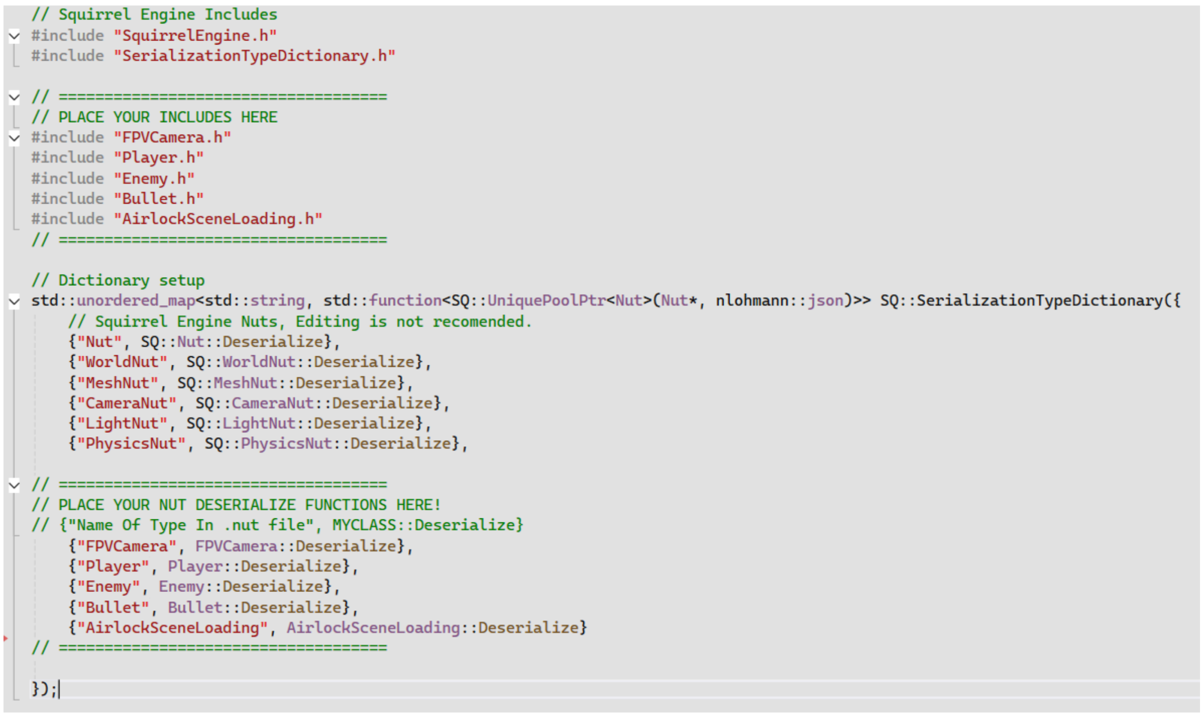
```cpp
// Check the player is inside the airlock area.
Player* player = GetTree()->GetRootNut()->GetNut<Player>("Scene/Main Player");
if (player && promise.get() == nullptr && LenSqrV3(player->GetGlobalPosition() - V3(0, 0, 20)) <= 25) {
    // If so start the scene loading on another thread
    promise = 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 8: 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 to call for this type.

The SerializationTypeDictionary is defined in the engine but needs to be initialized 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}
// ===================================

});
```

Figure 9: DeserializationTypeDictionary Initialization

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.

If the nut pointer is not null, the deserialization function should perform deserialization on that nut pointer instead.

The deserialization function should also call its parent class's function before doing any deserialization of parameters.

.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

Figure 10: Table of .shape and .mat JSON attributes

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. You should instead change the size of the collision shape
- ➔ Once added to the main scene tree a PhysicsNut should not be removed unless being deleted.
- ➔ A PhysicsNut requires a Collision Shape before it is added to the main scene tree or an error will occur.

Best Practice

Below is a list of best practice rules to follow:

- ➔ Specify all parameters of a nut in a .nut file, even if they are default. This keeps values clear and prevents uninitialized values.
- ➔ Prefer Shelled Nuts and .nut files over programmatic creation of nuts.
- ➔ Use camelCase for variable names, PascalCase for class and function names and SCREAMING_SNAKE_CASE for constants.
- ➔ Don’t delete a nut mid frame loop, instead use `QueueDestroy()`, this will tell the engine to safely delete the nut at the end of the frame.
- ➔ Don’t manage ownership to a nut with a raw pointer. UniquePoolPtr should always be used for nut ownership.
- ➔ Don’t reference resources with a raw pointer. Resources should always be referenced with shared pointers.
- ➔ Delete all objects in the Pool Allocation Service pool allocators before the application quits. To prevent this, always use a UniquePoolPtr. Failing to do so will result in destructors not being called and potential errors.

Error Codes

There are a total of 25 error codes which the engine can throw.

Error Code	Meaning	From Class(es)	Function
1	DirectX Math not Supported on CPU	GraphicsDX11	init
2	Failed to register window class	GraphicsDX11	init
3	Failed to create window	GraphicsDX11	init
4	Failed to create, DX11 Device, Device Context or Swap Chain	GraphicsDX11	init
5	Failed to get back buffer from swap chain	GraphicsDX11	init
6	Failed to create render target view	GraphicsDX11	init
7	Failed to create depth/stencil buffer	GraphicsDX11	init
8	Failed to create depth/stencil view	GraphicsDX11	init
9	Failed to create depth/stencil state	GraphicsDX11	init
10	Failed to create rasterizer state	GraphicsDX11	init
11	Options File failed to open	Entry	main
12	Options file has no window name	Entry	main
13	Options file has no window size	Entry	main
14	Options file window size formatted incorrectly	Entry	main
15	Options file has no on load nut	Entry	main
16	Shelled Nut being loaded but a nut inside it has no type	ShelledNut	Instantiate
17	Shelled Nut being loaded but a nut inside it has no name	Nut	Deserialize
18	Error Loading Mesh	Mesh	Load
19	Pool Allocation Service Unknown Error	PoolAllocationService	MakeUniquePoolPtr
20	Background colour not set	Entry	main
21	Background colour wrong format	Entry	main
22	Trying to allocate to a full pool.	PoolAllocator	New & Alloc
23	Target FPS not set	Entry	main
24	Pool block size too small or blocks not aligned. Make block size more than or equal to 8, and divisible by 8.	PoolAllocator	PoolAllocator
25	Physics Nut does not have a collision shape when ready called	PhysicsNut	Ready

Figure 11: Error Codes Table

CMP316 - Engine Development - Squirrel Engine

Critical Reflection

Cormac Somerville - 2200592

With my engine I was able to create a simple shooter game which demonstrated most of the core capabilities of the engine.
I was able to successfully integrate physics and deserialization into my engine. As well as have a 3D scene in which objects could be placed and manipulated.

There are various areas in which the engine shows strengths and weaknesses. I have sectioned these out below.

Pool Allocator

I was able to successfully implement and integrate a pool allocator (Gregory, 2018) into the engine and use it for storing all nuts. The pool allocator partially solves the issue of defragmentation as well as providing faster allocation and deallocation of memory, minimizing the performance impact of creating and deleting nuts as well as loading scenes. It also keeps objects semi continuous. One other success relating to pool allocators is with the pool allocation service. This automatically chooses the right size of pool to use and creates the required unique pointer and object automatically. It also features the ability to fall back onto new and delete if there is no pool big enough or if there is no capacity in the current pool.

Despite these benefits there are various improvements which could be made.
One would be expanding the use of the pool allocators inside the engine to cover resources. Resources take up larger blocks of memory and are also being loaded and unloaded at any time. Using the pool allocator for these would introduce the same benefits as it does for nuts. This would require shared and weak pointers for the pool allocators. There is already a partial implementation of these in the pool allocator header file, though they would need casting functions and new/delete support for fallback too.
One other improvement would be allowing the Pool Allocation Service to create new pools once one fills up rather than falling back on new and delete.

Overall I have learnt a lot from implementing pool allocators and it has allowed me to see the benefits of custom memory allocators in games development and their drawbacks.

3D Transformations

I believe overall the 3D transformations aspect of the engine has been a success. Inherited transformations work well with reparenting and editing parent positions. The use of quaternions has also helped refresh and improve my knowledge when it comes to 3D transformations.
One improvement could be to add global scale and rotation getters and setters for WorldNuts. These functions are currently missing and would give the developer more control over their objects.
Get Euler Angles is another common function missing from the current 3D transformations stack.
The missing global rotation functions create an issue with the physics integration where PhysicsNuts which are childrens of a rotated object won't behave as expected.

Physics Integration

Overall the physics integration has been a success. Collision detection is simple and elegant with physics simulation appearing seamless to the user without much setup.

I’ve learned a lot about library integration as JOLT (Rouwe, no date a) required a self build of the library and there were various parameters which needed to match between building of the lib and building of my application.
Physics simulation only happens on objects once they are in the main tree. I believe this is a reasonable limitation as creating and managing multiple physics worlds would be a complex task that most games would not require.
One issue is if a nut is removed from the tree and not deleted it still stays as an active body in the physics system, JOLT has functionality for deactivating a body but not removing it, this would add complexity but would be one solution to this problem.

Services

Services work as expected and provide all the necessary functionality within the engine one would expect.
I currently use a Service Locator pattern from (Nystrom, 2014) which the developer has to go through to access the underlying services of the engine. A benefit of the service locator pattern is to prevent errors from occurring when a developer tries to access a service which has not been setup or initialized yet.
In hindsight I would have used Singletons for my Services, it would have made access cleaner from the developer side.
Services are set up so that their construction and destruction order does not matter and any ordered initialization happens in the main function, in order. The only way the developer can access an uninitialized service is by editing critical engine code. A service locator provides the same level of global access as a singleton. Additionally, the use of singletons would have the added benefit of allowing files to only include the services they need rather than all services.

Abstraction

I believe abstraction is handled well in the engine. Each service which requires abstraction has been fully abstracted by the use of an abstract interface class. These services are Graphics, Physics and Input.
Mesh & Material resources also feature abstraction. Mesh requires storing references to DirectX buffers whereas material has a special storage format for usage in DirectX buffers. Originally Collision Shape was also abstracted but was later set to be universal as shapes were changed to being per nut rather than being shared, this was done as JOLT stores density as a shape parameter whereas I wanted density to be an object parameter.
Material could also not be abstracted and only have the material class, this would be slightly less performant as some copying of values would need to be done for every nut rendered but would mean the code is cleaner and load code does not need to be repeated when a new renderer is added.

Resources

Overall resources and resource management has been handled well. One improvement I would like to make would be the easy ability to make unique copies of resources at runtime or instantiation time, this would allow editing of a material colour without it affecting all materials which use the same file.

Deserialization

Overall I am very happy with my deserialization approach, it provides a simple and dynamic solution for string to type mapping. It makes use of a map of strings to functions to deserialize nuts dynamically at runtime. It also allows developers to write their own custom deserialization for their own nut types if they wish. It makes use of JSON (Lohmann, 2013) which is human readable and versatile. It also allows for the future addition of an editor to be easily added on top, as the editor could output .nut files.

One improvement could be the amount of code needed to be copied over as there is a small block of code which needs to be copied into every deserialization function written by the developer. I’ve tried to mitigate this by providing a search and replace friendly template. But the user also has to update the serialization type dictionary themselves.

One other improvement I would have liked to make would be to allow .nut files to be directly referenced in other .nut files allowing for perhaps a generic enemy.nut file which is referenced in main.nut multiple times and the enemy parameters can still be passed in like supplying a speed etc, but rather than having children it has a link to another nut file.

Final Remarks

In summary I am pleased with the engine I have been able to produce. I have been able to learn and use many core concepts of engine development and integrate them successfully. There is still room for improvement and changes but I don't believe these hinder the development of a potential game.

References

bvisness (2016) Handmade Math (v2.0.0) [C++]. https://github.com/HandmadeMath/HandmadeMath

De Vries, J. (2014) Light Casters Available at: https://learnopengl.com/Advanced-Lighting/Bloom (Accessed: 5 December 2024)

Godot Foundation (2007) Godot Accessible at: https://godotengine.org/ (Accessed On: 17 December 2024)

Gregory, J. (2018) Game engine architecture. 3rd edition. Massachusetts: A K Peters/CRC Press

Heesh, D. (2024) Doxygen v1.12.0 Accessible at: https://www.doxygen.nl/download.html (Accessed on: 17 December 2024)

Kulling, K (no date a) Assimp [C++] https://github.com/assimp/assimp (Accessed On: 13 December 2024)

Kulling, K (no date b) Importing Data Access by C++ class interface [C++] https://assimp-docs.readthedocs.io/en/latest/usage/use_the_lib.html#access-by-c-class-interface (Accessed On: 13 December 2024)

Lohmann N (2013) JSON (v3.11.3) [C++] https://github.com/nlohmann/json (Accessed On: 22 November 2024)

Microsoft (2009) DirectX 11 (v11.1) [C++] https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/

Microsoft (2019) Keyboard and Mouse Input Accessible at: https://learn.microsoft.com/en-us/windows/win32/inputdev/user-input (Accessed On: 8 November 2024)

Microsoft (2023) Taking Advantage of High-Definition Mouse Movement Accessible at: https://learn.microsoft.com/en-us/windows/win32/dxtecharts/taking-advantage-of-high-dpi-mouse-movement (Accessed On: 17 November 2024)

Nystrom, R. (2014) Game Programming Patterns Unknown: Genever Benning

Rouwe, J. (no date a) Jolt [C++] https://github.com/jrouwe/JoltPhysics (Accessed On: 14 December 2024)

Rouwe, J. (no date b) JoltPhysics HelloWorld.cpp https://github.com/jrouwe/JoltPhysics/blob/master/HelloWorld/HelloWorld.cpp (Accessed On: 14 December 2024)