Engine Class Structure

Mesh Collision

The triangles used to generate the collision mesh are read in from a binary file generated by the asset pipeline (see the pipeline docs for details.) These triangles are represented by 3 points in 3D space and are used to create geometric planes utilising the Directx toolkit's plane in their simple maths library. The plane, 3 points and what type of track it is (on track, off track, boost etc.) are stored in a MeshTri object. All of the MeshTri's are stored in a single vector, then they are split into a configurable sized grid for optimization. This grid only stores a pointer to the original MeshTri as to not waste memory. All of these are stored within a single Track object.

MeshTri contains the logic for calculating if a line intersect with the plane within the three points that define its triangle. The method DoesLineIntersect will take a start point, direction and angle limitation, then will return true if a collision is found, as well as populating a reference pointer to store the value of the triangle hit as well as the point of intersection. The algorithm used will determine if the collision is valid within the parameters passed, ensure the normal is facing the correct way to allow a collision, then it will calculate the line intersect. If the line intersect is within the distance passed, then it will use barycentric coordinates to calculate if the intersect lies between the three points on the plane which define the triangle. If any of the steps fail, it returns early to save on computation.

The Track object has a method with the same parameters and name as MeshTri (DoesLineIntersect), which uses the line of collision to calculate which segments of the collision grid it lies in. The method will then iterate through the grid checking each triangle for collision while keeping track of the closest intersection found. If any intersect is found, the closest values are stored in the reference pointers and true is returned.

Model Loading (SDKMeshGO3D)

The SDKMESH class was created for handling models within the game. The model loader utilised the "CreateFromSDKMESH" function from the DirectX Toolkit to load models in, and the DX helper library's "ReadData" functionality to validate that the model was of a valid header signature – this ability was particularly useful when switching from SDKMESH to SDKMESH2, to validate that the models being loaded had updated correctly. As well as loading in the model here, the material configurations are loaded to apply at render time.

To load a model as SDKMESH its materials must be pulled in to a PBREffectFactory to produce a PBREffect for each individual material. At runtime this PBREffect has its IBL materials set, and optionally, its albedo material for any animation overrides.

Animated materials are applied through the render function, where after switching to the model's resource heap the animation script checks for the current animation frame, sets that as the model's diffuse material, and then renders it. To make materials look metallic, IBL materials are switched to fix the DirectX ToolKit bug where RMA materials make no difference.

For memory management, a reset function is available to clear out loaded model data from resource descriptors and the SDKMESH object.

Track Physics

The logic for moving along the Track surface is contained within the TrackMagnet class. Each physics tick TrackMagnet objects check in a line beneath them for collisions with the Track's collision mesh. This is repeated in three other locations around the center of the object, this ensures it will not fall between small gaps in the mesh, such as material seams. If a collision is determined the object will rotate towards the normal of the surface found such that if the object is above the mesh, it would be rotated to sit flush against it by the time it had landed on it. Additionally, if the object is not in contact with the mesh, but in range for collision, then a local gravity force towards the track is applied.

Objects that are resting on the Track are continually rotated to stay on its surface and not clip through it. This is achieved by find the point of intersection directly under the object with the triangle its resting on, then finding a second point of intersection on that same triangle, but applying the object's velocity (or its world forward if velocity is 0) to the line of intersection. Using these two points a rotation vector can be generated by taking their difference. This difference combined with the normal of the intersected triangle is then used to create a new world transform for the object, which is correctly rotated on the Track mesh. This rotation is then lerped toward to smooth the movement and the object's velocity is adjusted to run parallel with the tracks surface to prevent clipping.

Wall collision is also contained within TrackMagnet and uses the same method of finding collisions with the collision mesh, but uses four vertical lines at the objects corners and four horizontal lines around the objects middle. If any of these checks find an intersection then the normal of the triangle hit is compared with the velocity of the object, if these values are diverging, then the collision is ignored as the object are already separating. If not, then the object's velocity is reflected in the triangles plane, then is adjusted to run parallel with the track to force the collision to resolve in 2D space (along the plane the object is currently moving across.) This collision also has an optional flag to dampen velocity on collision, this extra functionality will compare the velocity before and after the collision and take the angle between the two values. If this angle is high then the object has only slightly changed direction so velocity is only dampened slightly, whereas if the angle is low then the object has sharply changed direction, so velocity is greatly dampened.

Animations

The AnimationController class allows for objects to have multiple models with movement, scale and rotation animations without altering the hit-box or velocity of the object it is attached to. The controller stores a vector of AnimationModel objects which are added with a string to identify them. These models store an SDKMesh and will retain the original transform values of that mesh when it is created. The controller also has an std::map of std::vectors of AnimationModel pointers, which stores 'sets' of models. These sets are added by providing an identification string, which is used as key for the map, then a vector of strings which should be the identifying strings of the models that define which models should be included in the set. These sets can then be switched between with a single function call and the controller will only render its active set.

The controller is attached to another game object and will adjust all contained models to its transform, while also applying original transforms of each model. The controller can also take additional transform values which are used to actually create the animations. The position, rotation and scale values can be set directly, for instance to make a vehicle's wheels turn with its velocity. There is also a system of std::queue's for position and rotation, these values will be interpolated between based on a time value given. These queues can be set up for any movement or turning animation, such as spinning an object around.

Movement and Drifting

The controlled movement class is a component for all objects that move along a Track. The values for movement are configurable both in the toolkit and at runtime. This class can either take inputs from a user or it can be set to only take inputs from code, these inputs are used to set certain flags for movement: accelerate, decelerate, turn left/right and drift. These flags are processed every tick and will adjust the object's velocity accordingly. Drifting can be activated only when turning and will lock the object into a set turning arc which is sharper than turning and turning against the drift will lessen this angle. Once drifting ends, if the drift was long enough, the object's velocity is transformed to its world forward direction and a small boost is applied.

Gliding

In the player class there is a gliding state which is triggered by being in contact with a triangle tagged as “jump pad”. While in this state the players AnimationController switches to a gliding model set, gravity is lowered, drag is reduced, turn rate is lowered and acceleration is forced to maximum. The gliding state is only exited if the player comes into contact with the track again, or it gets respawned, which takes longer to trigger while gliding too.

Respawning

The player objects needs the ability to respawn, both because there can be sections of Track where falling off is possible and intended, but also because there is always some possibility that players will find themselves out of bounds. There are currently two conditions that will trigger a respawn, first if a player is not above any track for more than second and second if a player is on off-track tagged mesh for more than five seconds. The respawn location is determined by using the front of a std::queue of world transforms, this queue being the players most recent valid locations on the track. While respawning, the player cannot control their character, all velocity and state values are reset and the players object is lerped to the found position, plus a small offset upwards to prevent clipping.

Artificial Intelligence

The AI in the ThICC engine is a racing based AI split into two classes MoveAI and KartAI and all instances are managed by a single AIScheduler object. MoveAI contains the main bulk of logic that both classes use and is currently used on both players and certain items. The AI works by dynamically calculating a driving line at regular intervals determined by the AIScheduler. The line is created by projecting the objects transform forward on the track surface in small configurable intervals. If the projected position is off the track, then a projection is tried left and right to see if those will find a valid route, this process is recursive. The AI also ensures that any route tried does not diverge from the next waypoint, this prevents a route being found moving in the wrong direction. Each update tick the AI will check if its current velocity is moving towards the calculated racing line. If the current velocity deflect too far off the racing line, then the AI will turn its object until the deflection is back within allowed range. If the AI finds itself off the track for too long it will backtrack to the last position it was on the track and if the AI cannot find a route, it will head directly to the next waypoint in a straight line.

The Kart AI is a specialised child class of the move AI, it uses its parents logic for movement using the same driving line. The unique functionality of this AI is the ability to use items. In its current iteration item using logic is very basic, when an item is obtained the AI will wait a random amount of time between a minimum and maximum value, then use the item. If the item type is a speed boost, there is an extra requirement of having a minimum straight distance to the next waypoint in its driving line.

Box Collision

The PhysModel class stores an Orientated Bounding Box struct from the DirectX Toolkit. This struct needs to know its centre, it's extents (the distance from the center to each side) and it's orientation. These centre and extents are created from a json file storing the 8 corners of the physmodel mesh. The orientation is quaternion and is created from the PhysModel's orientation matrix decomposed into yaw pitch and roll which a quaternion is created from. These values are updated every update tick.

A Collision Manager singleton class has a static function called Collision Detection And Response which is called in the game scene's update and a vector of all PhysModels in the game scene are passed in. All of the physmodel's box colliders are then checked to be intersecting with it each other using DirectX's in-built OBB collision.For each intersection a collision struct is created and the two intersecting PhysModels are added to it as pointers as well as the normal of the collision if two players are intersecting. This struct is then pushed on to a vector and once all intersections have been checked this vector is iterated through. For each collision in the vector the first PhysModel is dynamically casted to a player pointer to check if it is a player. If it is a player then the second PhysModel is casted to a player to check the same thing. If so then a player response function is called which uses impulse resolution system to push the players apart. If it the second PhysModel isn't a player then it casted to an item box class. If it is an item box then the item box's collision function is called giving the colliding player an item. If it the second model is not an item box then it is casted to an explosion class (created from bombs) and if it is an explosion then the explosion classes collision response function is called. If the first PhysModel is not a player then then the second PhysModel is casted to player which if it is then the previous item box and explosion checks are performed on the first PhysModel.

A vector of all the items in the game scene are also passed into the Collision Detection And Response function and this vector and the PhysModel vector is passed into a Check Resolve Item Collisions function. This function iterates through the item vector first of all checking it is has a mesh (not all items have meshes). If it does have a mesh then the PhysModel vector is iterated through and each model is casted to a player to see if it is one. If it is a player then the player's and the item's mesh's box colliders are checking to be intersecting. If they are then then item's collision response function is called and the player is passed in. The PhysModel iteration is also broken out of once a intersection occurs. If no item and player intersection occurs then the item vector is checked against itself. If any item is intersecting with another then both are set to be destroyed in the game scene.

Items

All items in the game inherit from an abstract Item class. This class has two pure virtual functions: Hit By Player and Use. These functions are overloaded with the child item classes specific player response and use respectfully. The abstract item class also stores a pointer to a item mesh struct which holds a unique pointers for a mesh and an animation mesh. The mesh is an instantiation of track magnet and is where all the item's physics and collisions are applied. The animation mesh is the one that is rendered and is where the animations such as spinning are.

Item mesh structs are created and loaded when the game scene is loaded in item pools. This occurs in an item pools class which holds a map of item type enum to an item pool struct. This struct hold the size of the item pool and a queue of the item pool itself. This queue hold item mesh structs for one type of item. When an item is constructed it passes in the item type that it is into the item pool class through the service locator and the class returns an item mesh of that type and removes it from the item pool. If that pool is empty then a new item mesh is created and returned. When an item is destroyed the item mesh is reset and pushed on to the back of it's respective item pool queue. This also occurs when the game scene is unloaded, all items give back their meshes and the item pool deletes them all.

When a player uses an item it is initially stored as an item type enum before being spawned in the game scene. This occurs by the binding a create item function in the game scene into a std::function in the player class. When a player presses the use button this function is called and the item type the player currently has is passed in, this calls the function in the game scene which returns the newly created item as a pointer to the player class. If the created item has a mesh then it is pushed onto a trailing items vector otherwise the item's use function is just called. All the items in the trailing item vector are then set to appear behind the player for as long as the use button is held down or pressed again depending on the item. When the use button is released or pressed again the first item in the trailing item vector is popped of the vector and the item's use function is called. Also when items are created in the game scene they're added to another item vector which is used for item collision as well as being looped through during the game scene's update in order to call the update function on each item. When an item is flagged to be destroyed this is checked for the the game scene's update and when true the item is deleted and removed from the vector as well as giving it's item mesh back to the item pool.

Camera

The camera is handled in the Camera class. It stores a variable of the enumeration class CameraType. This type decides how the camera will behave. For the all the camera types it updates it position, rotation and the position it is looking at. Then it goes into a switch case in which any updates for specific cameras are executed. The values that camera uses to update are stored in the CameraData struct. The struct stores a variables like spin amount which is used for the orbit and independent camera. It also stores vectors of the field of views for each camera, the positions and look at positions for the cinematic camera.

Sound

The sounds are all stored in a vector in the audio manager. A json file stores the names and the attributes of all the sound such as the volume, pitch, pan and whether it loops. These are read in and given to the sound when it is created. The addToSoundsList function and pass in the sound with the tag and this gets put into the vector. The audio manager then is called throughout the code whenever a sound is needed. To play a sound, call the audio manager through the service locator and call the play function. It is the method to stop, pause and resume a sound.

Scene Manager

The various scenes within the ThICC engine are handled via a SceneManager class. The scene manager acts as an interface for the various scene objects within the game. It handles rendering and updating of 2D and 3D objects as well as loading and unloading different scenes. Unloading scenes consists of the “expensive” objects which would primarily be 3D objects such as the Track models, cameras, skyboxes, Item pools being reset or cleared from memory. The different scenes can also be added and set through the scene manager’s interface making for a wrapper to manage all the scenes from one location which can be accessed via the service locator. The expensive load and unload of scenes is done during switching from the current scene to a new scene which is done via a flag which checks to see every frame if a new scene has been requested. A 15 frame delay is placed between loading a new scene and unloading the previous scene to avoid memory issues. The DirectX garbage collector is utilised to free memory.