001

Tileset Generation System
Development and Design Document

Daniel George Mark Dore

Brief

In this project, I have created a tileset-based generation system to allow worlds to be generated via separate 'islands' with doorways connecting each one in order. Using a tree data structure to store the world elements, users can generate, clear and regenerate worlds at a call of a function, while a seed system allows users to create the same worlds multiple times via an integer.

Contents

List of Figures

Design

1 Introduction

A tileset 'is a grid of tiles used to create a game's layout' (), commonly used in 2D games to create their worlds. Usually, each tile is the same size and placed next to each other on a grid, with some tiles containing script to act as object such as platforms or killzones.

In my project, a tileset will be a collection of similar designed 'islands' that connect together via doorways or openings to create randomly designed worlds and levels. The generation of the tiles will be controlled through a single tile manager placed in the world. The user can also modify how long each branch of tiles will be, how many total tiles in the world and more. Tilesets are uncommon to see in 3D games, but they can be useful for procedural generation and randomness.

1.1 Warframe (Digital Extremes, 2013) uses a tileset generation system for all of it's non-open world levels - these being Plains of Eidalon, Orb Vallis and the surface of the Cambian Drift. Different tilesets allow for different environments to be created from the same system, while certain objective tiles - such as boss rooms or spy vaults - can be forced to spawn in the level.

2 Aims

The main aims of the project are:

- A system that can create a level from a chosen tileset
- Utilizes an n-tree for storage and query
- Simple to add to or modify
- Minimal amount of classes
- Easy setup, allowing levels to be cleared and created quickly
- Seeding, to minimize storage of the level to only player modifications

These aims will be achieved by:

2.1 Chosen Tileset - Different data tables will contain the different tileset information. For example, a space tileset will be kept solely in the Space data table, kept away from the forest tileset. This allows different doors and pathways to be used, rather than using a common static mesh across all tilesets.

2.2 n-Tree - A n-tree will be used to store all of the data of the tiles spawned in the world. Integars pointing to the index of each nodes parent and any children nodes currently existing, along with a pointer to the tile in the world.

2.3 Simply to modify - Designers will only need to change a few structs or add to enums to get their desired result

2.4 Minimal Classes - A limit of 5 parent classes has been set to make sure the system is as simple as possible

2.5 Easy Setup - Designers will only need to add one class to the world to have the whole system work properly

2.6 Seeding - Seeds will be used to reduce the amount of saving required to load a world after the player has exited. Additionally, seeding will allow for levels to be recreated over multiple playthroughs.

Development

1 Data Setup

Commit

I started by adding the main classes I will use in the system - these being the TileManager and Tile Actors and the TileDoorPosition ActorComponent. TileManager will hold the n-tree containing the world data - named GeneratedTree - along with spawning new tiles and connecting tiles together. Tile will be the parent class of all tiles placed in the world, with a BluePrint class child class being the parent these inherit from. TileDoorPosition will be the component designers add to the tiles to indicate where doors are, what type of door they are and their rotation.

Next, I added the FTileNode struct and the ETileDoorType enum. FTileNode is what the n-tree is made from, stored in an array for easy query. Currently, it contains a pointer to a Tile class, the integar of the parent node in the tree and a TMap of FName / int pairs. The FName designates the connection name relevant to that child. An additional contructor is added to the struct, using a Tile pointer and parent index as arguments. Originally, I was going to be pointers to other FTileNode structs, but UE5 doesn't allow this.

```
USTRUCT(BlueprintType)
struct TILESETGENERATION_API FTileNode
{
public:
	GENERATED_BODY();

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	class ATile* Tile;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	int Parent;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TMap<FName, int> Children;


	// Constructors / Destructors
	FTileNode();
	FTileNode(ATile* NewTile, int NewParent);
	~FTileNode();
};
```

ETileDoorType contains the selectable types of door in each tile. This will be used to stop incorrect doors from being connected to each other. Currently holds values of SmallDoor, MediumDoor and LargeDoor.

```
UENUM(BlueprintType, Category = "Tile")
enum ETileDoorType
{
	Small UMETA(DisplayName = "Small Door"),
	Medium UMETA(DisplayName = "Medium Door"),
	Large UMETA(DisplayName = "Large Door"),
};
```

Next, I added an FName and a ETileDoorType property to TileDoorPosition, used to hold the name and type of door. These are used by the Tile class, where it finds all TileDoorPositions and sorts them into two TMaps - one holding the class of the door position and one holding the type of door. I decided to use maps as they cannot have duplicate entries, so if the function to find all door positions runs an additional time for some reason, extra entries are not inputted. However, I could be updated to a struct and array if issues arise.

```
ATile::ATile(){
// Get all TileDoorPositions attached to this Tile
	TArray<UTileDoorPosition*> TDP;
	GetComponents<UTileDoorPosition>(TDP);

	// Then sort them into the maps
	for (UTileDoorPosition* i : TDP) {
		DoorPositions.Add(i->Name, i);
		Doors.Add(i->Name, i->Type);
	}
}
```

Finally, I setup the TileManager for next time. I started by adding an FObjectFinder to collect the data table asset of FTileData and store it for use in finding tiles. Different data tables will be able to be created for different tilesets - such as forest or desert themes.

```
USTRUCT(BlueprintType)
struct TILESETGENERATION_API FTileData : public FTableRowBase
{
public:
	GENERATED_BODY();

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
	TSubclassOf<class ATile> Class;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TArray<FName> Tags;

	// Constructors / Destructors
	FTileData();
	~FTileData();
};
```

Next, I added the array for the GeneratedTree and added a integar to the current node index. I also added

all of the functions that I want to implement in the TileManager as comments, followed by implementing the GenerateTileLevel function for use later.

2 Tag Query and Adding to the tree

Commit

I started by replacing the CurrentNode variable type from a FTileNode* to an integar, allowing me to search through the array successfully via index. Unlike regular c++, Unreal doesn't allow pointers to structs. Additionally, I added the VisibleAnywhere tag to the GeneratedTree property, allowing me to see the tree in the editor. Next, I implemented a second constructor in FTileNode, allowing creation of new structs with the arguments of ATile* and integar. This will be used in the AddTreeNode function.

```
FTileNode::FTileNode(ATile* NewTile, int NewParent)
{
	Tile = NewTile;
	Parent = NewParent;
}
```

AddTreeNode is simple - it checks if the parameter of NewTile is valid, before adding a new FTileNode to the tree with the CurrentNode as its parent, then adds to CurrentNode before returning true. If NewTile is invalid, then the function returns false.

```
bool ATileManager::AddTreeNode(ATile* NewTile)
{
	if (NewTile) {
		GeneratedTree.Add(FTileNode(NewTile,
		↪  CurrentNode));
		CurrentNode++;
		return true;
	}
	return false;
}
```

Next, I implemented the GetTileDoorPositions function in the Tile class - used to sort the door positions into the two maps talked about in Part One. It starts by checking if the TMaps are empty, followed by getting all components of class UTileDoorPoisitions and storing them in an array. Finally, for each item in that array new Pairs of data are added to the maps.

```
void ATile::GetTileDoorPositions()
{
	// Check if the maps are empty.  If they are, then...
	if (DoorPositions.Num() == 0) {
		// Get all TileDoorPositions attached to this
		↪  Tile
		TArray<UTileDoorPosition*> TDP;
		GetComponents<UTileDoorPosition>(TDP);

		UE_LOG(LogTemp, Warning,
		↪  TEXT("TileDoorPositions found : %i"),
		↪  TDP.Num());

		// Then sort them into the maps
		for (UTileDoorPosition* i : TDP) {
			DoorPositions.Add(i->Name, i);
			Doors.Add(i->Name, i->Type);
		}
	}
}
```

To allow the manager to search for a correct starting tile, I implemented the GetTileMatchingTag function. As each data table element has an array of FName tags, the function goes through each element in the data table and checks the tag arrays to see if it contains the inputted tag. If it does, it adds the class of the entry to a separate output array. Once all entries are queried, it generates a random int from 0 to the output arrays length, finally returning the Tile class in the randomly generated index.

```
TSubclassOf<ATile> ATileManager::GetTileMatchingTag(FName Tag)
{
	// Initialize a storing array
	TArray<TSubclassOf<ATile>> FoundTiles;

	// Check if the data table is found
	if (TileDataTable) {
		// Store all of the row names in the data
		↪  table, and initilize an array to store tags
		↪  found
		TArray<FName> RowNames =
		↪  TileDataTable->GetRowNames();
		TArray<FName> CurrentTags;

		// For each element in the data table, check
		↪  that its tag array contains the tag wanted.
		↪  If it does, add it to the FoundTiles array
		for (FName i : RowNames) {
			CurrentTags =
			↪  TileDataTable->FindRow<FTileData>(i,
			↪  "", false)->Tags;
			if (CurrentTags.Contains(Tag)) {
				FoundTiles.Add(TileDataTable->
			    FindRow<FTileData>
			    (i, "", false)->Class);
			}
		}
	}

	UE_LOG(LogTemp, Warning, TEXT("Amount Found: %i"),
	↪  FoundTiles.Num());

	// If at least one tag was found, get a random tile
	↪  found
	if (FoundTiles.Num() != 0) {
		return FoundTiles[FMath::RandRange(0,
		↪  FoundTiles.Num() - 1)];
```

```
        }

        // Else, return null
        return NULL;
}
```

If no matching tags are found, the function returns NULL instead. Finally, I started the GenerateTileLevel function by using GetTileMatchingTag to find a 'Spawn' tile and creating it in the world.

```
void ATileManager::GenerateTileLevel()
{
        ATile* NewestTile = nullptr;

        UE_LOG(LogTemp, Warning, TEXT("Generating"));

        if (TileDataTable) {
                // Start by finding a tile with the "Start" tag
                NewestTile = GetWorld()->SpawnActor<ATile>
            (GetTileMatchingTag(FName("Spawn")),
            FVector(0.0f, 0.0f, 0.0f), FRotator());
                AddTreeNode(NewestTile);
        }
}
```

As there is currently only one tile with the 'Spawn' tag GetTileMatchingTag successfully returns the Hanger tile each time, where it is then spawned in the world via GenerateTileLevel and added to the GeneratedTree via AddTreeNode.

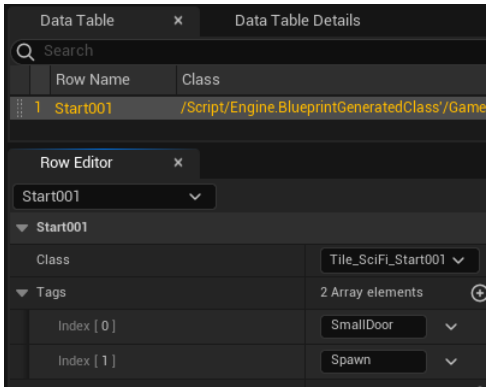

Figure 1. The Data Table - named DT TileData - as of v0.2. As shown, each entry can have an array of tags associated with it

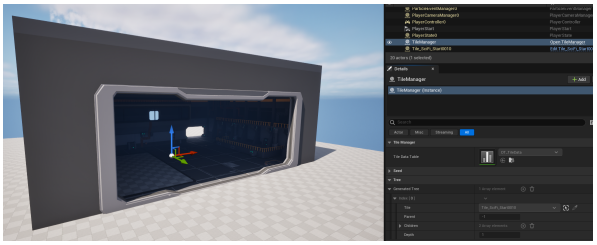

Figure 2. The Hanger spawned via GenerateTileLevel. The data of this tile has been added to the GeneratedTree

3 Parent Spawn Tile and Spawning Children

Commit

I started by updating GenerateTileLevel to spawn an amount of child tiles in the world based on how many TileDoorPositions the tile had. However, early on I found an issue - I couldn't easily convert the EDoorType to a FName to search for a tag - which I solved by implementing the GetTileMatchingDoor function. It works in a similar way to GetTileMatchingTag, but instead it queries a EDoorType array in each data table entry instead of the FName one. I also updated the TileData struct to contain this array.

```
USTRUCT(BlueprintType)
struct FTileData
        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        TArray<FName> Tags;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        TArray<TEnumAsByte<ETileDoorType>> Doors;
```

Another issue arose when I started to get the data from the Tile about each DoorPosition - you cannot get data from an TMap via an index, rather only via a key or value. This meant that I couldn't search for a matching EDoorType as I couldn't iterate over the map. My solution was to convert the two maps into one TArray of struct FDoorData.

```
USTRUCT(BlueprintType)
struct TILESETGENERATION_API FDoorData
{
public:
        GENERATED_BODY();

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
```

```
        FName Name;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        TEnumAsByte<ETileDoorType> Type;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        class UTileDoorPosition* Object;


        FDoorData();
        FDoorData(FName InName, TEnumAsByte<ETileDoorType>
        ↪  InType, UTileDoorPosition* InObject);
        ~FDoorData();
};
```

Next, I implemented the GetMatchingDoorPosition in the Tile class, which returns a matching door position based on a parameter. I added a randomness factor via choosing a random matching tile if it has multiple doors of the same type.

```
int ATile::GetMatchingDoorPosition(TEnumAsByte<ETileDoorType>
↪  InType)
{
int MatchingDoors = 0;

for (FDoorData i : Doors) {
        if (i.Type == InType) {
                MatchingDoors++;
        }
}

if (MatchingDoors > 0) {
        int ran = FMath::RandRange(0, MatchingDoors - 1);
        UE_LOG(LogTemp, Warning, TEXT("ran is equal to %i"),
        ↪  ran);
        return ran;
}
UE_LOG(LogTemp, Warning, TEXT("out of band"));
return 0;
}
```

I updated both the AddTreeNode and GenerateTileLevel functions. To make sure multiple tiles are not created for the same doorway when generating new child tiles, AddTreeNode now initializes the node's children array with invalid indices before adding to the tree. This allows GenerateTileLevel to check if there is a valid connection before spawning new tiles.

```
ATileManager::AddTileNode
FTileNode NewNode = FTileNode(NewTile, CurrentNode);
for (int i = 0; i < NewTile->Doors.Num(); i++) {
        NewNode.Children.Add(NewTile->Doors[i].Name, -1);
}

GeneratedTree.Add(NewNode);
return true;
```

Finally, I added the ability to spawn child tiles. GenerateTileLevel first spawns a tile with the Start tag as v0.2, with the additional effect of spawing a new child tile for each doorway in the parent, using GetMatchingDoorPositions to randomly choose each connector. To test this works properly, I added the new SmallCorridor tile.

```
ATileManager::GenerateTileLevel
for (int i = 0; i < ParentTile->Doors.Num(); i++) {
        // Find a tile with a door and spawn it
        NewestTile = GetWorld()->SpawnActor<ATile>
        (GetTileMatchingDoor(ParentTile->Doors[i].Type)
        , FVector(0.0f, 0.0f, 0.0f), FRotator());
        AddTreeNode(NewestTile);
        GeneratedTree[CurrentNode].Children.Add
        (ParentTile->Doors[0].Name,
        GeneratedTree.Num() - 1);

        // Get a random doorway in that tile that matches the
        ↪  current door type and connect it to the parent
        int DoorIndex = NewestTile->GetMatchingDoorPosition
        (ParentTile->Doors[i].Type);
        GeneratedTree[CurrentNode + i + 1].
        Children.Add(NewestTile->Doors[DoorIndex].Name,
        ↪  CurrentNode);
                }
```

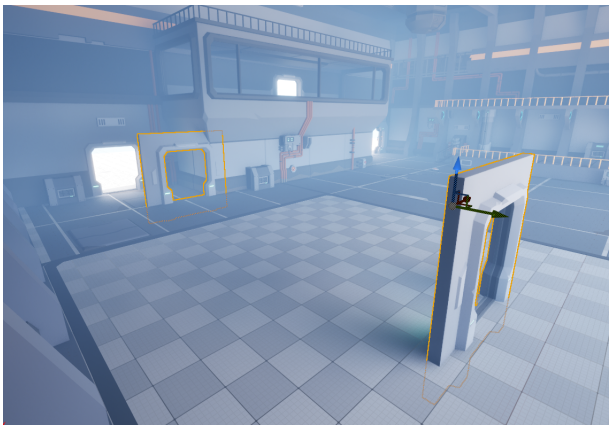

Figure 3. The spawn Hanger tile and the new SmallCorridor child tile spawned in the world. Either the north or south doorway will be chosen in SmallCorridor

4 Tile Rotation

Commit

In this session I implemented the child tile rotation to match the doorway of each tile together. I completed this by adding a rotation to the child tile based on the parent tile's door position rotation as well as

the rotation of the parent tile itself. The child tile is then rotated again based on the chosen door position's rotation.

```
// Rotate the new tile to match the doorway
NewestTile->AddActorLocalRotation(ParentTile->Doors[i].Object-
   >GetRelativeRotation() + ParentTile->GetActorRotation());
NewestTile->AddActorLocalRotation(NewestTile->Doors[DoorIndex].
   Object->GetRelativeRotation());
```

ParentTile-GetActorRotation was added to allow root tiles to have any rotation if needed. Finally, an offset is added to the tile to move it into the correct position.

```
// Finaly, move the new tile
NewestTile->AddActorWorldOffset(ParentTile->Doors[i].Object-
   >GetComponentLocation() -
   NewestTile->Doors[DoorIndex].Object->
   GetComponentLocation());
```

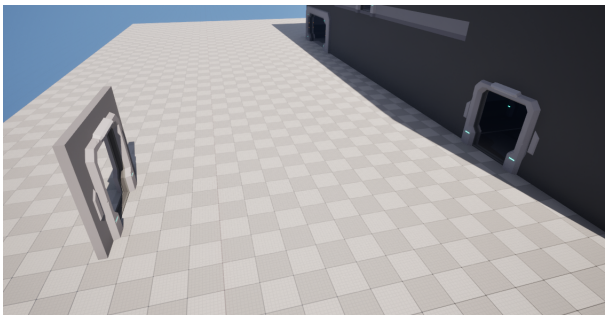

Figure 4. The SmallCorridor child tile has been rotated to properly face the door of the Hanger tile.

5 Child Rework

Commit

In this session, I reworked the generation of child tiles slightly and modified how children are stored in the tree via a new struct. Once again, as you can't easily iterate over each node in a TMap. Additionally, to query what door type I needed to find for a new tile, I had to find the child tile in the tree then query each door type. To offset this, I converted the Children property of the TileNode struct from a TMap of TPair FName / int to a TArray of FTileChildren - who has properties of FName and int from the TMap as well as a ETileDoorType property.

```
USTRUCT(BlueprintType)
struct TILESETGENERATION_API FTileChildren
{
public:
        GENERATED_BODY();

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FName DoorName;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        TEnumAsByte<ETileDoorType> Type;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int IndexToTile;


        // Constructors / Destructors
        FTileChildren();
        FTileChildren(FName InName, TEnumAsByte<ETileDoorType>
        ↪  InType, int InIndex);
        ~FTileChildren();
};
```

Next, I moved the child spawn script from the GenerateTileLevel function to it's own GenerateNode function. The function uses the same code, with the additional arguments of DoorType and ChildIndex.

```
void ATileManager::GenerateNode(TEnumAsByte<ETileDoorType>
↪  DoorType, int ChildIndex)
```

Finally, I updated the child generation portion of GenerateTileLevet to include all doorways for each tile via a nested for loop. To make sure the code doesn't generate tiles for doorways already filled, it queries each doorways IndexToTile to check if it is invalid before calling GenerateNode.

```
void ATileManager::GenerateTileLevel
CurrentBranchLength++;

// For each node in the tree
for (int i = 0; i < GeneratedTree.Num(); i++) { //
        CurrentNode = i;
        // For each child node of node i
        for (int j = 0; j < GeneratedTree[i].Children.Num();
        ↪  j++) {
                // Generate a new tile
                if (GeneratedTree[i].Children[j].IndexToTile ==
                ↪  -1) {
                        GenerateNode(GeneratedTree[i].Children[j].Type,
                        ↪  j);
                }
        }
}
```

6 Max Branch Length

Commit

In this session, I implemented the MaxBranchLength limitation. I started by adding a new property to FTileNode - that being Depth or 'the length of the path from the root to that node' (geeksforgeeks). If a new node's depth exceeds MaxBranchLength, the node isn't created.

```
void ATileManager::GenerateNode
{
if ((GeneratedTree[CurrentNode].Depth + 1) <= MaxBranchLength)
↪  {
        ATile* ChildTile = GetWorld()->SpawnActor<ATile>
    (GetTileMatchingDoor(DoorType), FVector(0.0f, 0.0f, 0.0f),
    ↪  FRotator());
        AddTreeNode(ChildTile);
 ...
}
```

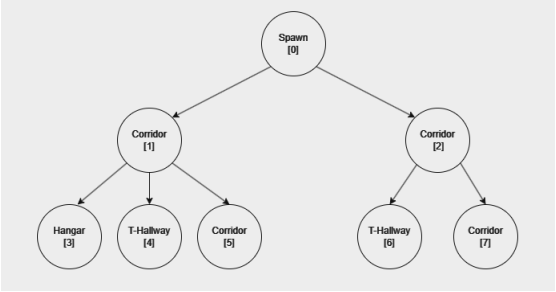

Figure 5. A diagram of a tree data structure. A nodes depth is simply how many connections are needed to pass over before reaching the node

7 Clear and Regenerate

Commit

In this session, I implemented the ClearTileLevel and RegenerateTileLevel functions. ClearTileLevel iterates across each node of the tree backwards, destroying the tile object stored in each node. Once completed, the garbage is collected to clear unneeded memory space and then the tree is emptied.

RegenerateTileLevel simply calls ClearTileLevel followed by GenerateTileLevel.

```
void ATileManager::ClearTileLevel()
{
if (GeneratedTree.Num() != 0) {
        for (int i = GeneratedTree.Num() - 1; i > -1; i--) {
                GeneratedTree[i].Tile->Destroy();
        }

        // Garbage collect
        GEngine->ForceGarbageCollection();

        // Clear the tree
        GeneratedTree.Empty();
}
}

void ATileManager::RegenerateTileLevel()
{
        ClearTileLevel();
        GenerateTileLevel();
}
```

8 Dot and Cross

Commit

In this session, I reworked how child tiles are rotated into place. Previously, tiles would be rotated to match the rotation of the parent's door position as well as the offset of the parent tile itself. Then the child tile would be rotated again based on the rotation of the chosen child doorway.

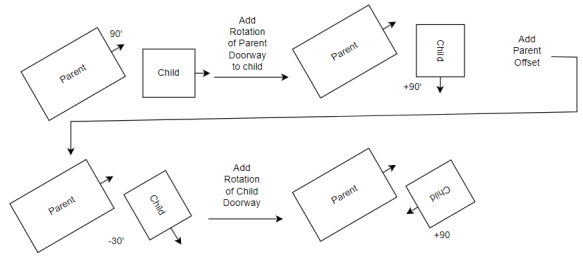

Figure 6. A diagram how the tiles rotated previously. As the tiles only ever used doorways on 90 degrees intervals, the rotation worked.

However, this method doesn't work with tiles not on 90 degree intervals. If a doorway had a different offset - say 30 degrees - the tile would rotate inaccurately as shown via the diagram below.

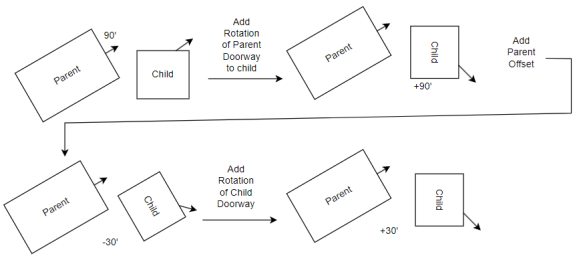

Figure 7. A diagram how the tiles rotated previously. If the tiles used a 30 degree offset the rotation would be inaccurate

I fixed this by finding the dot product between the two doorways that needed connecting, then calculating the cross product to gather the direction they needed to rotate. Finally, the tiles are rotated 180 degrees so that the child tile is facing the door followed by adding the calculated angle as an offset.

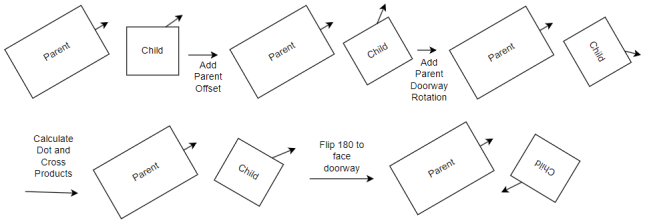

Figure 8. A diagram how the tiles rotated now. Calculating the dot product finds the angle needed to rotate, while the cross product takes this angle and finds the direction

```
void ATileManager::GenerateNode
// Add the parent tile's offset to the new tile
ChildTile->AddActorLocalRotation(GeneratedTree[CurrentNode]
    .Tile->GetActorRotation());
ChildTile->AddActorLocalRotation(GeneratedTree[CurrentNode]
    .Tile->Doors[ChildIndex].Object->GetRelativeRotation());

// Find the dot product between the child tile doorway and the
↪  parent tile doorway
FVector VecA =
↪  GeneratedTree[CurrentNode].Tile->Doors[ChildIndex].
    Object->GetForwardVector();
VecA.Normalize();

FVector VecB =
↪  ChildTile->Doors[DoorIndex].Object->GetForwardVector();
VecB.Normalize();

float OutAngle =
↪  FMath::RadiansToDegrees(acosf(FVector::DotProduct(VecA,
↪  VecB)));

// Calculate the cross product to determin what direction to
↪  rotate in
if (FVector::CrossProduct(VecA, VecB).Z > 0)
{
        OutAngle = -OutAngle;
}

// Rotate the tile so it is now facing the door
ChildTile->AddActorLocalRotation(FRotator(0.0f, 180.0f, 0.0f));
ChildTile->AddActorLocalRotation(FRotator(0.0f, OutAngle,
↪  0.0f));
```

9 Seeding

Commit

In this session, I implemented seed world generation through Unreal's FRandomStream. When GenerateTileLevel is called, a random integar is chosen between 0 and 2000, which is then used to create a new SeedStream.

```
void ATileManager::GenerateTileLevel()
// Make a new random seed
CurrentSeed = FMath::RandRange(0, 2000);
SeedStream = FRandomStream(CurrentSeed);
```

When a tile is selected in either GetTileMatchingTag or GetTileMatchingDoor, instead of generating a random int from range via FMath, it instead uses SeedStream. This uses the seed value to generate the same number every time when the same seed is used.

```
TSubclassOf<ATile>
↪  ATileManager::GetTileMatchingDoor(TEnumAsByte<ETileDoorType>)
return FoundTiles[FMath::RandRange(0, FoundTiles.Num() - 1)];
// to...
return FoundTiles[SeedStream.RandRange(0, FoundTiles.Num() -
↪  1)];
```

10 Choose Seed

Commit

In this session, I implemented the ability to generate a tile level from a inputted seed. I conducted this by adding a new function GenerateTileLevelFromSeed, which uses an integar parameter to generate a seed for the seed stream before starting generating. Additionally, GenerateTileLevel was updated to take in an boolean argument to check if a new seed required creating before starting level generation.

```
void ATileManager::GenerateTileLevel(bool bGenerateNewSeed)
if (bGenerateNewSeed) {
                // Make a new random seed
                CurrentSeed = FMath::RandRange(0, 2000);
                SeedStream = FRandomStream(CurrentSeed);
        }

 void ATileManager::GenerateTileLevelFromSeed(int Seed)
{
        SeedStream = FRandomStream(Seed);
        GenerateTileLevel(false);
}
```

This allows users to input their own seed values, instead of having different seeds generated each time.

11 Tree Settings

Commit

In this session, I added the ability to get and change the tree settings via two new functions - GetTreeSettings and SetTreeSettings. Due to functions only able to return one data type, I created a new struct to hold all tree settings data and updated GenerateTileLevel and GenerateNode to use this struct instead.

```
FTreeSettings ATileManager::GetTreeSettings()
{
        return TreeSettings;
}

void ATileManager::SetTreeSettings(FTreeSettings NewSettings)
{
        TreeSettings = NewSettings;
}
```

12 Tree Queries

Commit

In this session, I added the ability to query the tree via either an index or ATile pointer to return either the tile of that index or the index of that tile pointer respectively. FindTileFromIndex simply checks if the index argument is in range of the GeneratedTree's length, then returns the pointer at that index.

```
ATile* ATileManager::FindTileFromIndex(int IndexToFind)
{
        if (IndexToFind > -1 && IndexToFind <
        ↪  (GeneratedTree.Num())) {
                return GeneratedTree[IndexToFind].Tile;
        }
        return nullptr;
}
```

FindIndexFromTile iterates over the entire array from the root, comparing the pointer at that index to the pointer argument. If them match, it returns the index, else if all array elements are queried with no matches then an invalid index is returned.

```
int ATileManager::FindIndexFromTile(ATile* TileToFind)
{
        for (int i = 0; i < GeneratedTree.Num() - 1; i++) {
                if (GeneratedTree[i].Tile == TileToFind) {
                        return i;
                }
        }
        return -1;
}
```

Further Improvements

I still think there are areas that I could improve the project. These include, but are not limited to:

1 Template GetTile functions - Instead of using two separate functions to do similar tasks with different parameters, I could create a template function that checks the parameter type before searching either the tag or door arrays as they do currently. This would introduce the issue of incorrect data types being used as arguments though.

2 Link seed generation to system clock - Linking the random seed generation when no seed is used to the system clock would allow more random seeds to be generated, instead of the limit of just 2000 different seeds. This is simple to do, as Unreal has the function Now() in FDateTime.

3 Replacing FTileDoorPositions with structs - Tiles currently use ActorComponents to designate the location and direction of doorways on tiles, these could be updated to use structs instead to store data. However, this would require a small rework to the tile rotation code, as currently the TileDoorPosition's forward vector is used.

Usage

1 Adding new Door Types

To add a new door type, open the header file for Core/Data/TileDoorType and add a new enum element to the list. Compile your project, then you can choose the new door type.

```
#include "CoreMinimal.h"

UENUM(BlueprintType, Category = "Tile")
enum ETileDoorType
{
    Small UMETA(DisplayName = "Small Door"),
    Medium UMETA(DisplayName = "Medium Door"),
    Large UMETA(DisplayName = "Large Door"),
};
```

Figure 9. The TileDoorType header file, where you can add new elements to the enum.

2 Creating new tiles

To create a new tile, create a class tile of BP-Tile Blueprint class in Content/Core/Tileset. You can then create your tile with static or skeletal meshes.

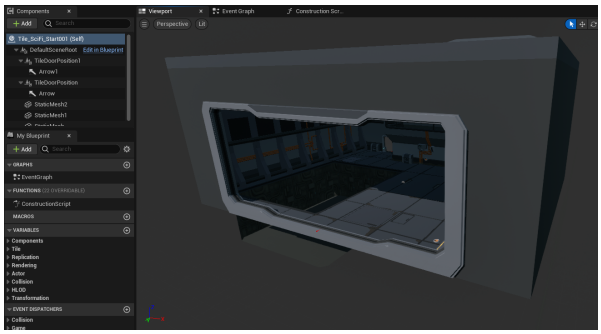

Figure 10. A example tile. You can insert your meshes or classes as components to create your tile.

To add doorways, add the TileDoorPosition actor component to your new tile, then attack an arrow component to it to see its rotation easier. Position the component in its correct location and rotation with the arrow facing away from the center. You can then set the doorway name and type in Details/Tile Door Position.

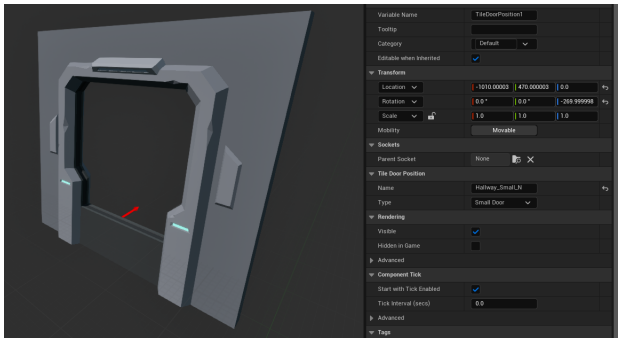

Figure 11. A TileDoorPosition. The ArrowComponent is attached to the door position to see its rotation and location on the tile.

3 Creating a tileset

To create a new tileset, you can create a new data table of struct . You can then insert new entries to the table, each containing an individual tile, as well as any tags you want to give to the tile. Make sure to also insert the correct door types.

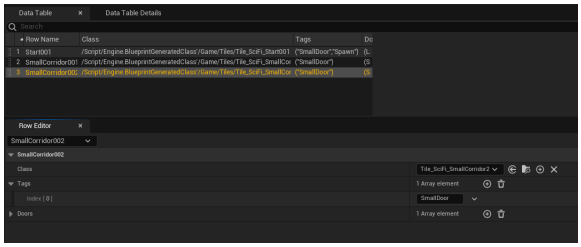

Figure 12. A data table for a tileset. Each element is a separate tile, with the tags and door types used by the tile.

You can then use this data table in the TileManager instead of the default one collected via an FObjectFinder.

Bibliography

References

Epic Games. (2014) Unreal Engine 4/5 [Game Engine]

Godot. (2018) 'Using Tilesets', Available at: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilesets.html (Accessed: 27 December 2023)

Digital Extremes. (2013) Warframe [Game]

GeeksForGeeks. 'Introduction to Tree – Data Structure and Algorithm Tutorials', Available at: https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/ (Accessed: 27 December 2023)

Imaver. (May 2023) 'Dot and Cross Product in Unreal Engine 4', Available at: https://forums.unrealengine.com/t/how-to-get-an-angle-between-2-vectors/280850/39 (Accessed: 27 December 2023)