001

Prototype: FoodGame
Development and Design Document

Daniel George Mark Dore

March 2023 - Present

Brief

Inspired by the fast-paced action of Crazy Taxi, the player is given their own fast food restaurant to pilot and maintain, while endless rounds of customers come and bombard them with orders. Every time the player completes an order, they are rewarded with extra round time and money and completing enough orders in a round freezes remaining seconds for use in the next. A simple control scheme keeps the game easy to learn, while the amount of different orders enhances replayability.

Will you keep up with the demand, or will your dream restaurant burn down in flames? Will you stick too the book, or will you give customers extra to keep them happy?

Contents

List of Figures

Design

Will you keep up with the demand, or will your dream restaurant burn down in flames?

1 Overview

Prototype: Food Game is fast-paced cooking game that combines elements of arcade games such as Crazy Taxi (Sega, 1999) with simulation games such as Cooking Simulator (Big Cheese Studio, 2019). The player is given their own restaurant to command, with multiple different styles with different features, where they must cook and deliver orders to customers on time.

The game is played in a first-person perspective, similarly to how other simulator games play, but a third-person camera will be provided if certain gameplay elements require it. The controls are also kept simple to use and understand - where left mouse button picks up or places items in the world and right mouse button interacts.

The time limit is split into two, with one being a global 'round' timer displaying how long the player has before they fail. They can add seconds too the clock by completing orders in a good time or with good accuracy and pause it once enough orders have been completed. Each order handed to the player has its own separate time limit, where failing to complete an order on time rewards no benefits.

2 Level Design

In the alpha version of the game, two levels will be available to the player - the tutorial level and the drive-thru. The tutorial level will be based off of a simulation, where new managers learn the ropes of the game before being given the real deal. The drive-thru level will be a simple fast-food restaurant with a drive thru, but wont be in use until later.

Further on in the games development, the levels will be randomly generated out of the asset parts, giving the player a different experience on each play through. Additionally, each level base will have a core element that sets them apart from the other bases such as having a drive-thru - where drive-thru customers have a shorter time limit than walk-ins - or being situated in the city center - where lunch rushes will be heavier but give more time to prep.

3 Cooking

Creating food is simple - the player just needs to combine multiple food items together in different ways to create recipes. However as the player only has access to one interact key, the systems for modifying items was designed to be free, lacking menus and encourage experimentation.

Some recipes, such as burgers, require the player to stack items on top of each other in a specific order. With stacking rules recipes can be set to have items in specific or general places. An example of this is requiring the bottom burger bun to be the start of the stack, while the burger patty can be anywhere between the bounds.

Crafting is kept free and menu-less with specific stations such as chopping boards providing ways of completing tasks. Instead of using a menu to chop food, all players need to do is place a knife and the item(s) they want to chop on a chopping board. Interacting with the board then completes the chopping task, providing the player with the output if a recipe is found.

4 Customers

Customers will be somewhat basic in the beginning where they would have a data table of items they can order, choose one item from the list, find a table, start a timer and wait for their food to arrive. If the food arrives before the timer completes, they judge the order before rewarding the player.

When judging an order, the AI will award the supplied item points in three separate categories: Accuracy (how accurate their food is to the item they ordered), Speed (how fast they recieved the order) and Presentation (how accurate items are stacked or positioned). The higher each score, the better the rewards are. Once all items are given to the customer (if multiple items are ordered), then the player is rewarded for the order.

For completing an order on time the player is given

a base amount of time added back to the round clock, which is increased by the amount of points they scored from the order. They are also awarded money, which is added to their total score at the end of each round. If a round is failed with money remaining, they can spend it to buy back in. Failing a certain amount of orders per round will cause the player to lose.

5 Difficulty

Each round completed will add a new challenge to the player, with differences in rewards and difficulty. Some challenges are rarer than others, with challenges being placed into separate tiers based on how many rounds complete are needed for them to appear. Challenges can also have a round limit.

Challenges could be as simple as adding a new menu item (like chips as a side for a burger), adding an option to an existing menu item (adding cheese or sauce to a burger) or making items cheaper for a round (so less money is rewarded to the player). They can also change the flow of the game entirely, such as adding a new customer type who favours presentation over speed, or one who has a shorter temper.

The player will also be able to spend some of the money they have accumulated over the rounds to clear an existing challenge, or pay a fee to avoid adding any new challenges.

Development

1 Player Character Iteration One

Commit

To start off, I implemented the first iteration of my player character. In this iteration, the player has access to two hands that can be activated by pressing or holding either of the mouse buttons - the left button activates the left hand and vice-versa. I implemented a press or hold system for activation, so pressing the key activates the hand until it is pressed again while holding the key activates it until it is released.

```
// --- Arms
void APlayerCharacter::LeftArmPressed()
{
	bLeftArmPressed = true;
	if (LeftArmState != EArmState::Pressed) {
		// Could use lambda here, but this will be used more than once
		ToggleEnableArm(LeftHandCollision, true);

		GetWorld()->GetTimerManager().ClearTimer(LeftArmHandle);
		GetWorld()->GetTimerManager().SetTimer(LeftArmHandle, FTimerDelegate::CreateUObject(this, &APlayerCharacter::LeftArmTimer), ArmPressTime, false, ArmPressTime);
	}
	else {
		// Disable and set arm state to Disabled
		LeftArmState = EArmState::Disabled;
		ToggleEnableArm(LeftHandCollision, true);
	}
}

void APlayerCharacter::LeftArmReleased()
{
	bLeftArmPressed = false;
	if (LeftArmState == EArmState::Held) {
		// Disable and set arm state to Disabled
		LeftArmState = EArmState::Disabled;
		ToggleEnableArm(LeftHandCollision, true);
	}
}

void APlayerCharacter::LeftArmTimer()
{
	if (bLeftArmPressed) {
		// We are holding, set state to Held
		UE_LOG(LogTemp, Warning, TEXT("Held"));
		LeftArmState = EArmState::Held;
	}
	else {
		// We have pressed, set state to Pressed
		UE_LOG(LogTemp, Warning, TEXT("Pressed"));
		LeftArmState = EArmState::Pressed;
	}
}
```

When a hand is activated, it turns on collision with items. Once an empty hand overlaps with an item in the world, it attaches to it until the hand is deactivated. In future development, each item would have had a boolean to designate if it was a two-handed item or not. Two hands allows the player to hold two single handed items at once, or one two handed item.

Video

While I liked the press and hold mechanic of the hands, the overlapping felt very clunky to maneuver items around the world to the players liking. It was also very difficult to aim your hands to choose which item to collect and if an item was out of the players reach, it was impossible to pick up (see video). In the next iteration, I will be implementing the ability to see what you are picking up before activating the hands.

2 Player Character Iteration Two

Commit

Iteration Two's main goal was to fix the accuracy problem that iteration one introduced. I started by adding a collision sphere to the player themselves, which adds a pickup to an array whenever they overlap. The size of this sphere can be modified via the InteractRange float.

```
// Called every frame
void APlayerCharacter::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

	if (InteractablesInRange.Num() != 0) {
		// Trace interactables infront of the player
		// Generate information for the trace
		FHitResult TraceHit = FHitResult(ForceInit);
		FVector TraceStart =
↪  FirstPersonCamera->GetComponentLocation() +
↪  (FirstPersonCamera->GetForwardVector() * 50);
		FVector TraceEnd = (TraceStart +
↪  (FirstPersonCamera->GetForwardVector() * InteractRange));
		ECollisionChannel TraceChannel =
↪  ECC_GameTraceChannel1;

		FCollisionQueryParams
↪  TraceParams(FName(TEXT("Interact Trace")), true, NULL);
		TraceParams.bTraceComplex = true;
		TraceParams.bReturnPhysicalMaterial = true;

		bool bInteractTrace =
↪  GetWorld()->LineTraceSingleByChannel(TraceHit, TraceStart,
↪  TraceEnd, TraceChannel, TraceParams);
		DrawDebugLine(GetWorld(), TraceStart, TraceEnd,
↪  FColor::Green, false, 5.f, ECC_GameTraceChannel1, 1.f);
		if (bInteractTrace) {
			if
↪  (TraceHit.Actor->IsA(AParentItem::StaticClass())) {
				// Set ItemContextWidget to
↪  HitLocation, enable player tracking
				// Also set
↪  InteractableLookingAt to OtherActor
				InteractableLookingAt =
↪  TraceHit.Actor.Get();
			}
			else {
				InteractableLookingAt =
↪  nullptr;
			}
		}
	}
```

While there is at least one item in the interact range, a trace is fired on tick which reacts to Interactables (items and future item variations). If this trace hits an item, a pointer is set to the item hit. Activating an arm while this pointer is not nullptr picks up the item, attaching it to the corresponding hand with the same press or hold method from iteration one. Deactivating the hand drops the item as before.

Before implementing this trace on tick I conducted research to find how expensive tracing was. According to a forum post on the Unreal Forums, with line tracing you 'can have hundreds (or even thousands maybe) traces per tick with minimal performance cost.' (TK-Master, Unreal Forums). However, using shape traces increased performance loss significantly. Additionally, reducing unneeded traces via comparisons can reduce the loss even further.

I prefer this method of picking up the items to iteration one. As stated before, iteration one did not allow the player to accurately choose what item they wanted to pick up unless said item was alone. With the new trace system, this problem is reduced but a new problem has arisen - tracing on tick can use a significant amount of memory if not handled well. However, this system can further be improved. In the next iteration, I will be deprecating the two-hand system for a single hand one, which will allow items to have a primary (pickup) and secondary (use) action via the mouse.

3 Player Character Iteration Three

Commit

Iteration Three's main goal was to allow for a primary/secondary action system, and a more accurate drop mode. As we no longer have two hands, anything that references an arm state has either been deprecated and removed or renamed. This included the press and hold system for both arms.

To implement the drop mode, I added a boolean variable to determine if the player is collecting or placing an item, named *bDropMode*. I also added a static mesh component to show where the item held will be placed in the world. While in the drop mode, the player can simply press the left mouse button again to place it instantly where they are looking, or hold the button down to use this 'hologram' to accurately place it in the world through a separate trace.

```
APlayerCharacter::Tick...
else if (PlacingMesh->IsVisible()) {
		// Trace to see where to place the hologram
		TraceStart =
↪  FirstPersonCamera->GetComponentLocation() +
↪  (FirstPersonCamera->GetForwardVector() * 50);
```

```
                TraceEnd = (TraceStart +
↪  (FirstPersonCamera->GetForwardVector() * InteractRange));
                TraceChannel = ECC_GameTraceChannel1;
                FVector dropLoc;

                FCollisionQueryParams
↪  TraceParams(FName(TEXT("Drop Trace")), true, NULL);
                TraceParams.bTraceComplex = true;
                TraceParams.bReturnPhysicalMaterial = true;

                bTrace =
↪  GetWorld()->LineTraceSingleByChannel(TraceHit, TraceStart,
↪  TraceEnd, TraceChannel, TraceParams);
                DrawDebugLine(GetWorld(), TraceStart, TraceEnd,
↪  FColor::Green, false, 5.f, ECC_GameTraceChannel1, 1.f);
                if (bTrace) {
                        PlacingMesh->SetWorldLocation(TraceHit.Location);
                }
                else {
                        PlacingMesh->SetWorldLocation(TraceEnd);
                }
        }
```

As the right hand has been removed, so has the size of the player's 'inventory'. To mitigate this, players can pick up multiple items instead of one, limited by each items weight. If the player tries to pick up an item that would exceed their weight, or if they are in the drop mode, nothing happens and the item is left in the world.

```
bool APlayerCharacter::CheckCanCollectItem(float NewItemWeight)
{
        if (CurrentWeight + NewItemWeight > MaxWeight) {
                UE_LOG(LogTemp, Warning, TEXT("Cant Collect"));
                return false;
        }
        else {
                UE_LOG(LogTemp, Warning, TEXT("Can Collect"));
                return true;
        }
}
```

This method of item interaction is the strongest one implemented so far. Allowing the player to still be creative with which items they collect at once, while allowing for accuracy with collecting and placing. It also allows for context or special actions to be implemented in the future as the right mouse button is unused. There are some areas for improvements, such as the tick event looking extremely messy which can introduce bugs down the line - separating each tick into their own functions could be a solution

4 Re-placing Items

Commit

This addition was a simple change - renaming and adding the ability to change which item the player is going to place. Anything named *Drop* is renamed to *Place* to better reflect what it is doing.

The player can change the active item they are going to place through the scroll wheel, with comparisons to make sure they cannot go out of the array bounds.

```
void APlayerCharacter::ChangeItem(float AxisValue)
{
        if (AxisValue == 1.0f) {
                if (HeldItems.Num() == 0) {
                        // Do nothing
                }
                else if (HeldItems.Num() == 1) {
                        CurrentHeldItem = 0;
                }
                else if (CurrentHeldItem + 1 >=
↪  HeldItems.Num()) {
                        CurrentHeldItem = 0;
                }
                else {
                        CurrentHeldItem++;
                }
        }
        else if (AxisValue == -1.0f) {
                if (HeldItems.Num() == 0) {
                        // Do nothing
                }
                else if (HeldItems.Num() == 1) {
                        CurrentHeldItem = 0;
                }
                else if (CurrentHeldItem - 1 <= -1) {
                        CurrentHeldItem = 0;
                }
                else {
                        CurrentHeldItem--;
                }
        }
}
```

5 Stations

Commit

Stations are one of the crafting methods that players can use to modify or change their items. Each station has their own crafting table that is stored as a data table. The first station that I implemented was a chopping board.

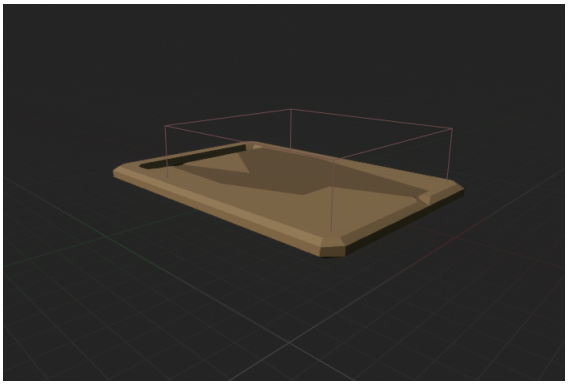

Figure 1. The Chopping Board Station

Stations have a context item slot, which only allows certain, specified items to be placed in. They also attach themselves to the station directly, but can still be taken from the station by picking them up. Before an item is placed on the station, they are checked for a match by calling GetStationSlot.

```
FItemSlot AParentStation::GetStationSlot(FString Name)
{
	int slotIndex = -1;

	// Check each item slot in the array for the input
	for (int i = 0; i < ItemSlots.Num(); i++) {
		for (int j = 0; j <
↪  ItemSlots[i].AcceptedItems.Num(); j++) {
			if (ItemSlots[i].AcceptedItems[j] ==
↪  Name) {
				return ItemSlots[i];
			}
			else if (ItemSlots[i].AcceptedItems[j]
↪  == "items_all") {
				slotIndex = i;
			}
		}
	}

	// If an example of "items_all" was found, then return
↪  that slot
	if (slotIndex != -1) {
		return ItemSlots[slotIndex];
	}
	// Else, return an empty slot
	else {
		UE_LOG(LogTemp, Warning, TEXT("Empty"));
		return FItemSlot{};
	}

}
```

After a match is found, the item or hologram (depending on if the player is holding or placing the item at the time) is moved and rotated to match the item slot's transform.

```
FTransform AParentStation::GetSlotTransform(FString SlotName)
{
	for (int i = 0; i < ItemSlots.Num(); i++) {
		if (ItemSlots[i].Slot == SlotName) {
			return ItemSlots[i].Transform;
		}
	}
	return FTransform{};
}
```

I also simplified the tick traces by moving them to their own separate functions - PlaceTrace and InteractTrace. For recipes, I added a new struct in the ItemDataLibaray called FRecipe. A recipe has properties for station type, what context item a recipe needs, what input items are needed and what output items are needed.

```
USTRUCT(BlueprintType)
struct FRecipe : public FTableRowBase
{
	GENERATED_USTRUCT_BODY();

public:
	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		FString ID;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		TEnumAsByte<EStationType> Station;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		FString ContextItem;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		TArray<FString> InputItems;

	UPROPERTY(EditAnywhere, BlueprintReadOnly)
		TArray<FString> OutputItems;

};
```

Video

I think the current iteration of chopping crafting via the chopping board station is good, but as the chopping board is small and usually gets moved around in an actual kitchen it should be treated more like an item rather than an in-place object. This can be easily remedied by having ParentStation extend from ParentItem instead of AActor as it does.

I also think the context item slot attaching is slightly restrictive, due to how the attachment works. If a station is at a different rotation that 0,0,0, the item does not match it position or rotation properly.

6 Main Menu Beginnings and Recipe

Commit

I began the work on the main menu level, creating the start of a building resembling a British-style drive-through McDonalds. Further on in development, more buildings will be added, including a restaurant in a busy mall, a town center fast food place and a higher class pub. These buildings will be used when the player chooses a level to play, with the exterior also being modified by what recipe or food they choose to start with.

I also added a function for searching for recipes in the data table. When a new item is overlapped in a stations crafting range, all ID's of the items currently in the range are taken and queried, to find if there is a matching recipe on the board. If there is, the recipe ID is stored until it is needed by the player. If no recipe is found, the FString is just set to empty.

```
void AParentStation::FindRecipe()
{
// Matching Recipes
TArray<FRecipe> FoundRecipes;
// Get all ID's in the crafting range
TArray<FString> InputItemIDs;

// Iterate across the station recipes for one matching
for (int i = 0; i < StationRecipes.Num() - 1; i++) {
        //if (StationRecipes[i].ContextItem == )
for (int i = 0; i < ItemsInCraftingRange.Num(); i++) {
        InputItemIDs.Add(ItemsInCraftingRange[i]->GetItemName());
}

// Iterate for matching recipe
TArray<FName> RecipeRows = Recipes->GetRowNames();
bool bRecipeFound = false;

TArray<FString> iids;
TArray<FString> frid;

FRecipe* cr;

return FRecipe();
// Iterate through all the recipes
for (int j = 0; j < RecipeRows.Num(); j++) {
// If the recipe hasen't been found, continue looping
if (!bRecipeFound) {
cr = Recipes->FindRow<FRecipe>(RecipeRows[j], "", false);
// Check if the recipe matches the station type
if (cr->Station == StationType) {
// Check if this recipe has the same context item
if (cr->ContextItem == ContextItemSlot.ItemInSlot) {
// Check if the recipe has the same amount of items used in
↪  crafting
if (cr->InputItems.Num() == InputItemIDs.Num()) {
iids = InputItemIDs; frid = cr->InputItems;
for (int k = 0; k < iids.Num(); k++) {
if (frid.Contains(iids[k])) {
frid.RemoveAt(k);  iids.RemoveAt(k);
}
}
if (iids.Num() == 0 && frid.Num() == 0)
{
// This is the item
bRecipeFound = true;
CurrentRecipes = *cr;
}}}}}}
if (!bRecipeFound) {
CurrentRecipes = {};
}
```

I like this approach to crafting, which feels more free and easier to use as it is not at all reliant on UI menus. As the recipe is only updated when the items in the range are updated, it shouldn't be very process heavy, even with the amount of queries.

7 Stations Rework

Commit

This addition was but somewhat complex - changing how stations extend from AActor to ParentItem instead. This was completed by creating a new project, importing all classes except Station, creating a subclass AParentStation from AParentItem, importing all of the script from the previous project to the new one, then swapping the new project for the old one. I attempted to just change ParentStation from *extends AActor* to *extends AParentItem*, but errors started to appear. This allows Stations to be picked up like items now.

I also took the time to better create a folder structure, while renaming files to better suit their need and use.

8 Crafting

Commit

To implement crafting, I first needed to add a way for the player to actually interact with the station. This is done through the secondary action. As the player only needs to press the button once to chop an item, I implemented a trace that checks if it hit a ParentStation which then casts to the object if it is. Before conducting any interaction, it also checks to see if a recipe has been set, and rejects any interaction if one hasn't.

```
void APlayerCharacter::SecondaryActionPress()
{
        TraceStart = FirstPersonCamera->GetComponentLocation()
↪  + (FirstPersonCamera->GetForwardVector() * 50);
```

```
        TraceEnd = (TraceStart +
↪  (FirstPersonCamera->GetForwardVector() * InteractRange));
        TraceChannel = ECC_GameTraceChannel2;
        FVector placeLoc;

        FCollisionQueryParams TraceParams(FName(TEXT("Drop
↪  Trace")), true, NULL);
        TraceParams.bTraceComplex = true;
        TraceParams.bReturnPhysicalMaterial = true;

        bTrace = GetWorld()->LineTraceSingleByChannel(TraceHit,
↪  TraceStart, TraceEnd, TraceChannel, TraceParams);
        if (bTrace) {
                if
↪  (TraceHit.Actor->IsA(AParentStation::StaticClass())) {
                        LastHitStation =
↪  Cast<AParentStation>(TraceHit.Actor.Get());
                        if (LastHitStation->CurrentRecipes.ID
↪  != "") {
                                LastHitStation->CraftRecipe();
                        }
                }
        }
}
```

Next was actually crafting an item. Before spawning any new items, I make it so crafting check to see if any item in the crafting range can be change to an output item. For example, in a recipe with an input of a tomato and outputs of two tomato slices - where all items are of the AParentItem class - the tomato input will be changed to a tomato slice before any new items are spawned.

```
void AParentStation::CraftRecipe()
{
TArray<FString> newItems = CurrentRecipes.OutputItems;

// Change any items currently in the range to the new items
for (int i = 0; i < ItemsInCraftingRange.Num(); i++) {
        ItemsInCraftingRange[i]->SetupItem(*Items->FindRow
                <FItemData>(FName(*newItems[0]), "", false));
        newItems.RemoveAt(0);
        }
}
```

After this, it checks to see if there are any remaining items needed to be spawned. If there are, it simply spawned them.

```
AParentStation::CraftItem...
// For any remaining items in newItems, create a new item class
        for (int j = 0; j < newItems.Num(); j++) {
                UE_LOG(LogTemp, Warning, TEXT("NewItem"));
                FItemData* newData =
↪  Items->FindRow<FItemData>(FName(*newItems[0]), "", false);
                AParentItem* nI =
↪  GetWorld()->SpawnActor<AParentItem>(newData->Class,
↪  CraftingRange->GetComponentLocation() + FVector(0.0f, 0.0f,
↪  30.0f), FRotator{});
                nI->SetupItem(*newData);
        }
```

Video

I like how the crafting is currently set up, but there are areas I want to add too. This includes - randomly spawning new items in the crafting range, modifying spawning code just in case other classes are used and introduce a need to hold down interaction for a specified time, instead of having crafting finish immediately.

I also want to add to the free-crafting system by removing the context item attachment just require the item to be placed in the crafting range too. This will remove the ability to take the context item with the station if the station is picked up, but the context item will act more like actual items rather than a new tool.

9 Free Crafting

Commit

As stated in the previous addition, I wanted to improve the free-crafting feeling by removing the context item attachment point and have stations just detect if a context item was added to the item range. This was simply added by checking any new items added against the context items allowed and adding it to a FString if it matched.

```
void AParentStation::OnCRBeginOverlap(UPrimitiveComponent*
↪  OverlappedComp, AActor* OtherActor, UPrimitiveComponent*
↪  OtherComp, int32 OtherBodyIndex, bool bFromSweep, const
↪  FHitResult& SweepResult)
{
        bool bAddItem = false;

        if (OtherActor->IsA(AParentItem::StaticClass())) {
                ItemsInCraftingRange.Add(Cast<AParentItem>(OtherActor));
                FindRecipe();
                AParentItem* collideItem =
↪  Cast<AParentItem>(OtherActor);
                // Check if it is a context item.
                for (int i = 0; i <
↪  ContextItemSlot.AcceptedItems.Num(); i++) {
                        if (collideItem->Data.ID ==
↪  ContextItemSlot.AcceptedItems[i])
                        {
                                AddContextItem(collideItem->Data.ID);
                                bAddItem = true;
                                break;
                        }
                }
                // Else
                if (bAddItem == false) {
                        ItemsInCraftingRange.Add(collideItem);
                        FindRecipe();
                }
        }
}
```

Additionally, to mitigate items spawning in each other and pinging off far away, I made any new items spawned by crafting to be placed in a random location inside the stations crafting range.

```
AParentStation::CraftRecipe...
AParentItem* nI =
↪  GetWorld()->SpawnActor<AParentItem>(newData->Class,
↪  UKismetMathLibrary::RandomPointInBoundingBox(CraftingRange->
    GetComponentLocation(),
    CraftingRange->GetUnscaledBoxExtent()), FRotator{});
```

Crafting only requires small changes to work correctly now, such as making sure other item classes (such as stations) can be spawned instead of items and making sure all ParentItems are changed successfully.

10 More Free Crafting

Commit

This addition should remedy the issues with crafting that were stated previously, these being - other sub classes can be spawned and changing successfully. Fixing changing is simply repeating items in the loop until all newItems are checked, while spawning other classes is completed by using the class provided from the data table row.

```
//AParentStation::CraftRecipe...
// Change any items currently in the range to the new items
for (int i = 0; i < ItemsInCraftingRange.Num(); i++) {
        ItemsInCraftingRange[i]->SetupItem
    (*Items->FindRow<FItemData>(FName(*newItems[0])
    , "", false));
        newItems.RemoveAt(0);
        if (Items->FindRow<FItemData>(FName(*newItems[j]), "",
↪  false)->Class == ItemsInCraftingRange[i]->Data.Class){
                ItemsInCraftingRange[i]->SetupItem
    (*Items->FindRow<FItemData>(FName(*newItems[j]), "",
↪  false));
                newItems.RemoveAt(j);
        }
        else {
                i--;
                j++;
        }

}

// For any remaining items in newItems, or any items requiring
↪  different classes, create a new item class
for (int j = 0; j < newItems.Num(); j++) {
for (j = 0; j < newItems.Num(); j++) {
        FItemData* newData =
↪  Items->FindRow<FItemData>(FName(*newItems[0]), "", false);
        AParentItem* nI =
↪  GetWorld()->SpawnActor<AParentItem>(newData->Class,

↪  UKismetMathLibrary::RandomPointInBoundingBox(CraftingRange->
    GetComponentLocation(),
    CraftingRange->GetUnscaledBoxExtent()), FRotator{});
```

In the video, a test recipe was created to make sure other classes could be spawned, this will be removed later in development.

Video

11 Plating

Commit

Implementing plating was next, allowing the player to actually serve food to a customer later on in development. Plates are child classes of ParentItem, allowing them to be interacted with and picked up like other items. However, they have additional properties, such as bDirty, or functionality, such as having the ability to allow other items to be attached to the plate. To plate, a new comparison is added to PlaceTrace, to see if a hit item is of class APlate.

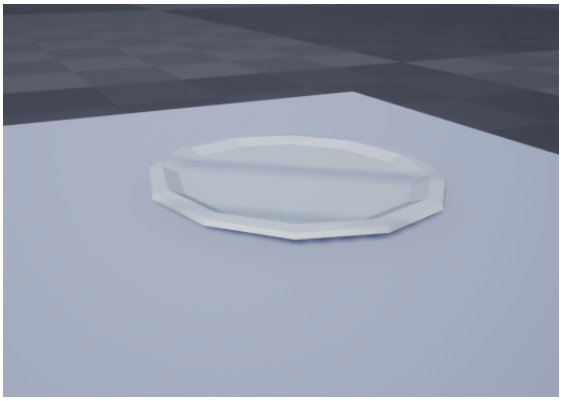

Figure 2. A Plate Object

```
// APlayerCharacter::PlaceItem...
if (TraceHit.Actor->IsA(APlate::StaticClass())){
        // Cast to the hit plate
        APlate* HitPlate = Cast<APlate>(TraceHit.Actor);

        // De-attach from the character
        HeldItems[0]->SetActorLocation(TraceHit.Location);
        HeldItems[0]->SetActorRotation(GetActorRotation() +
↪  FRotator(0.0f, 90.0f, 0.0f));
        CurrentWeight = CurrentWeight -
↪  HeldItems[0]->GetItemWeight();

        // Attach to the plate
        HitPlate->AttachedItems.Add(HeldItems[0]);
        HeldItems[0]->AttachToActor(HitPlate,
↪  FAttachmentTransformRules::KeepWorldTransform);
```

```
        HeldItems[0]->AttachedTo = HitPlate;
        HeldItems.RemoveAt(CurrentHeldItem);
}
else {
        // If it hits something, place it at the hit location
        HeldItems[0]->ToggleItemCollision(true);
        HeldItems[0]->DetachFromActor
        (FDetachmentTransformRules::KeepWorldTransform);
        HeldItems[0]->SetActorLocation(TraceHit.Location);
        HeldItems[0]->SetActorRotation(GetActorRotation() +
↪  FRotator(0.0f, 90.0f, 0.0f));
        CurrentWeight = CurrentWeight -
↪  HeldItems[0]->GetItemWeight();
        HeldItems.RemoveAt(CurrentHeldItem);
```

Items also have a pointer to any item they are attached too, allowing easy detaching later.

Video

12 Cooking

Commit

Cooking was the next feature I implemented, which was another way players could 'craft' new items from old ones. Before creating the cooker class, I reworked the recipe system by splitting each station type's recipes into separate structs and data tables. This simplifies the recipe system through removing unneeded checks when recipes are searched for - FindRecipe no longer needs to check for a matching enum.

```
// AParentStation::FindRecipe...
cr = Recipes->FindRow<FRecipe_Chop>(RecipeRows[j], "", false);
 // Check if this recipe has the same context item
 if (cr->ContextItem == ContextItemSlot.ItemInSlot) {
  // Check if the recipe has the same amount of items used in
↪  crafting
  if (cr->InputItems.Num() == InputItemIDs.Num()) {
   iids = InputItemIDs; frid = cr->InputItems;
   for (int k = 0; k < iids.Num(); k++) {
        if (frid.Contains(iids[k])) {
        frid.RemoveAt(k);  iids.RemoveAt(k);
        }
}
```

Additionally, cooking recipes only have one input and output, so there is no need for arrays. I also renamed the previous recipe struct to better suit how it is used.

```
struct FRecipe -> FRecipe_Chop

USTRUCT(BlueprintType)
struct FRecipe_Cook : public FTableRowBase
{
        GENERATED_USTRUCT_BODY();

public:
        UPROPERTY(EditAnywhere, BlueprintReadOnly)
                FString ID;

        UPROPERTY(EditAnywhere, BlueprintReadOnly)
                FString InputItems;

        UPROPERTY(EditAnywhere, BlueprintReadOnly)
                float CookTime;

        UPROPERTY(EditAnywhere, BlueprintReadOnly)
                FString OutputItems;
};
```

Following this, I added the components and setup the constructor in the AParentCooker class. Finally, I created a new material for when an item is left on/in a cooker for too long - named BurntMaterial.

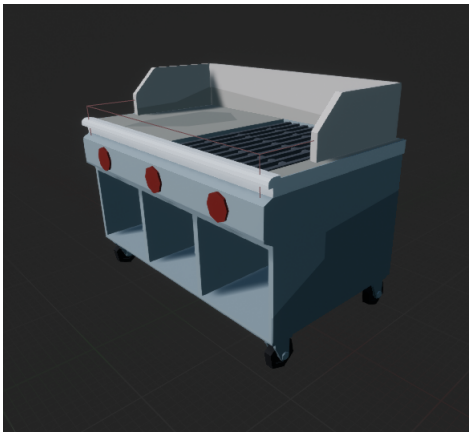

Figure 3. A Grill Object

13 Cooking Part Two

Commit

When an item without a recipe is cooked, it should burn instead. Additionally, some items shouldn't be able to be cooked - such as knifes, chopping boards and certain foodstuffs. I added a boolean in ItemData that allows the user to set which items can be cooked - bBurnable - as well as a second boolean in the Item itself which states if the item has burnt.

To allow players to actually cook their items, I added three functions to the Item class: StartCooking - which checks if the item is burnable - StopCooking - which pauses the timer if there is one active - and EndCooking - the timer has finished and the item should either become one from a recipe or burn.

```
void AParentItem::StartCooking(AParentCooker* Cooker,  float
↪  CookingTime)
{
// Set the AttachedActor to the cooker, but only if it's not
↪  attached to anything else
if (AttachedTo != nullptr && Data.bBurnable == true) {
        AttachedCooker = Cooker;

// Check if the timer exists
if
↪  (GetWorld()->GetTimerManager().TimerExists(CookingTimerHandle))
↪  {
        // If it doesn't, create it
        GetWorld()->GetTimerManager().SetTimer(CookingTimerHandle,
↪  FTimerDelegate::CreateUObject(this,
↪  &AParentItem::EndCooking), CookingTime, false,
↪  CookingTime);
}
```

```
else {
        // If it does, resume it
        GetWorld()->GetTimerManager().UnPauseTimer
    (CookingTimerHandle);
}
}
}

void AParentItem::StopCooking()
{
if (AttachedCooker != nullptr) {
// Dereference the pointer
AttachedCooker = nullptr;

// Pause the timer
GetWorld()->GetTimerManager().PauseTimer(CookingTimerHandle);
}
}

void AParentItem::EndCooking()
{
// Change the item to the recipe
AttachedCooker->OnCookingEnd(this);
}
```

StartCooking is called when an ParentItem object overlaps the cooker objects cooking range, with StopCooking called when the overlap ends. OnCookingEnd is the function that changes the item or burns it - by searching through it's recipe data table for a matching item.

```
void AParentCooker::OnCookingEnd(AParentItem* Item)
{
// Find the recipe related with this item.  If one is not found,
↪  then burn the item
FRecipe_Cook foundRecipe = FindRecipe(Item->Data.Name);
if (foundRecipe.ID == "") {
        Item->ItemMesh->SetMaterial(0, BurntMaterial);
        Item->bCannotCook = true;
}
// Else, set the item to the one in the recipe
else {
        Item->SetupItem(*Items->FindRow<FItemData>
     (FName(*foundRecipe.OutputItems), "", false));
}
}
```

Finally, I updated the cooking recipe struct with a enum to designate what cooker type is used per recipe - you shouldn't be able to cook a pizza on a grill, for example.

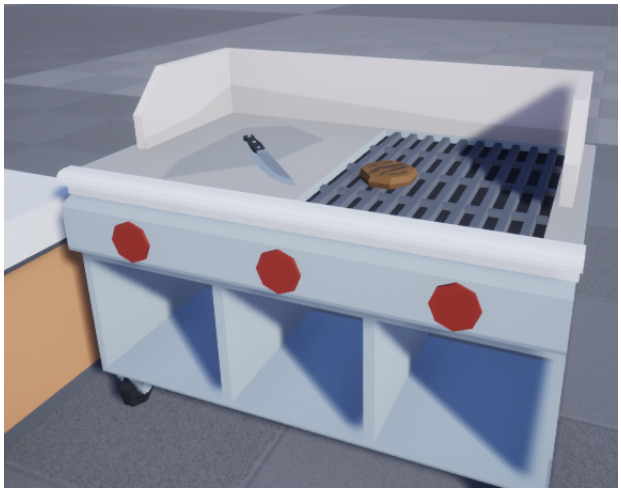

Figure 4. Knifes and utensils shouldn't be able to burn, while foodstuffs such as the burger should

I like how cooking is currently coming along - using timers on the individual items instead of the cooker itself reduces coding complexity. It also allows for objects to be cooked at different times based on the recipe.

14 Cooking Fixes

Commit

Some fixes were needed to complete the cooking implementation. This includes adding a default cooking time if FindRecipe returned empty, fixing a bug in StartCooking due to a misplaced comparison sign, adding dynamic binds to the cooking range and adding a check to restart cooking after a recipe has been completed.

```
//AParentCooker::OnCookingEnd...
// Check if the new item can be cooked.  If so, restart cooking
if (Item->Data.bBurnable == true) {
        Item->GetWorld()->GetTimerManager().ClearTimer
     (Item->CookingTimerHandle);
        Item->StartCooking(this,
     FindRecipe(Item->Data.ID).CookTime);
}
```

15 Binning Items

Commit

Next, I added a way to delete items from the game. Later on, a filter will need to be added to the bin to make sure important items can't be binned.

Figure 5. The bin allows the player to delete items they don't need

16 Hologram Material

Commit

To make sure the player knows that the hologram is actually the hologram and not just another item they can pick up floating in front of them, I added a translucent material to the PlacerMesh when set.

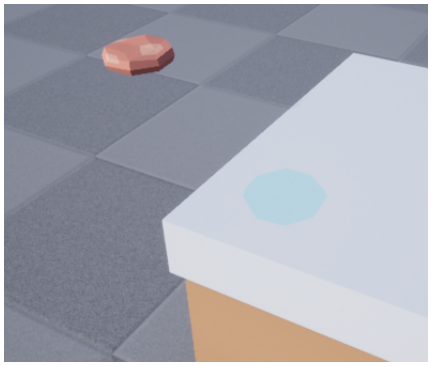

Figure 6. A example of the hologram material in use

I also fixed a bug with cooking recipes, where OnCookingEnd was searching with the items name and not its ID.

17 Cooking Bugfix

Commit

During the previous commits I somehow introduced a bug that would crash the game when an item was crafted, requiring a small rework on how items are changed before new items are spawned in CraftRecipe. The main change was to see if int j exceeded newItems.Num(), when true would stop the for loop.

```
// AParentStation::CraftRecipe...
if (j < newItems.Num()) {
 if (Items->FindRow<FItemData>(FName(*newItems[j]), "", false)
 ->Class == iICR[i]->Data.Class){
 iICR[i]->SetupItem(*Items->FindRow<FItemData>
 (FName(*newItems[j]), "", false));
 newItems.RemoveAt(j);
 iICR.RemoveAt(i);
 }
else {
 i--;
 j++;
}
```

I also modified the PlaceItem trace to feature a check if items are in a stack or are stack-able, while also moving default placement code to their own AttachedAt function - which needs a rename.

```
void APlayerCharacter::AttachAt(FVector Location)
{
HeldItems[0]->ToggleItemCollision(true);
HeldItems[0]->DetachFromActor(FDetachmentTransformRules
 ::KeepWorldTransform);
HeldItems[0]->SetActorLocation(Location);
HeldItems[0]->SetActorRotation(GetActorRotation() +
↪  FRotator(0.0f, 90.0f, 0.0f));
CurrentWeight = CurrentWeight - HeldItems[0]->GetItemWeight();
HeldItems.RemoveAt(CurrentHeldItem);
}
```

18 Stacking Objects

Commit

Next was allowing items to be stacked on top of each other and storing how items are stacked, in a similar way to how plates are stacked. To store them in the correct order, a TSet is used as it keeps its order when new indexes are added. Additionally, AttachedItems are moved from AParentItem to APlate, as only the plate uses this variable.

I also implemented DetachStack, which detaches all items above an item when picked up. For example, say

you have a stack of bun bottom, burger, lettuce leaf and bun top. Removing the burger will detach the bun top and lettuce leaf. If you collected the leaf instead, only the bun top would detach.

```
// Go through the Set from the last index, detaching items
bool bMatchingItem = false;
TArray<AParentItem*> sIA = StackedItems.Array();

for (int i = StackedItems.Num() - 1; i > 0; i--) {
        // Check if the items dont match and the item hasn't
↪   been found
        if (sIA[i] == ItemToDetachFrom ) {
                //  If the items are the same, then set
↪   bMatchingItem to true then remove it
                bMatchingItem = true;
                sIA[i]->DetachFromActor(FDetachmentTransformRules
    ::KeepWorldTransform);
                sIA[i]->ToggleItemCollision(true);
                sIA[i]->AttachedTo = nullptr;
                StackedItems.Remove(sIA[i]);
        }
        // Else, check if the item has been found already.  If
↪   it hasn't, remove this item
        else if (!bMatchingItem)
        {
                sIA[i]->DetachFromActor(FDetachmentTransformRules
    ::KeepWorldTransform);
                sIA[i]->ToggleItemCollision(true);
                sIA[i]->AttachedTo = nullptr;
                StackedItems.Remove(sIA[i]);
        }
}
```

Finally, the PlaceTrace is modified to check what class of the AttachedTo item is.

I like how stacking has been set up, allowing the player to freely create any stacks of items they want - with limitations. It is also enjoyable to see all of the items pop off together when picking up an item midway through the stack, while the ones below stay glued together. I plan to use the way the stack is stored to manage customer orders further on in development.

19 Stacking Bugfixes

Commit

When implementing item stacking, I forgot to implement the bStackable check, which accidentally allowed non-stackable items (such as knifes) to be added to stacks.

20 Containers

Commit

I started by removing ticking events from classes that currently do not use Tick for anything, improving performance. These classes include - ParentItem, Plate, ChoppingBoard and more.

Figure 7. Containers will be used to store cooked / prepared items, or deliver new ones

I also added the ParentContainer class. Containers are a child of ParentItem, which allow certain items to be stored inside of them. Only one item can be stored at a time, but a large amount of that item can be stored at once. Players can interact with an empty hand to collect an item from the container, or with a matching item in hand to store it.

```
bool AParentContainer::AddItemToContainer(AParentItem*
↪   ItemToAdd)
{
// Check if the container has an item stored already.
   // If it doesn't check if the new item ID can be stored
if (ContainedItem.ID == "" || Amount == 0) {
        if (AcceptedItems.Contains(ItemToAdd->Data.ID)) {
                // If the ID can be stored, set it to be this
↪   containers item
                SetupContainer(ItemToAdd->Data, 1);
                return true;
        }
}
// Check if the item is the same ID as the item stored
else if (ItemToAdd->Data.ID == ContainedItem.ID) {
        Amount++;
        return true;
}
return false;
}

bool AParentContainer::RemoveItemFromContainer(class
↪   APlayerCharacter CharacterToGiveItem)
{
        if (Amount > 0) {
                // Spawn
        }
```

```
        if (true) { return true; };
        return false;
}
```

21 Single Item Inventory and Weight System

Commit

I moved away from having a multi slot item 'inventory' to a single item one for simplification. Players are not really ever going to pick up multiple items at once. This also allows for placing with a single button without toggling between modes and reduces tick count when in range with a pickup while holding an item. Any reference to HeldItems was changed to HeldItem and the TArray of pointers was changed to a single one. I also deprecated and removed the weight system from the inventory, as the only limit to what the player can pick up will be the single slot.

I also fixed a bug with placing items - if the place trace hits air, it now places the item at the end of the trace rather than the world origin.

```
APlayerCharacter::PlaceItem...
}
// Else, drop it in mid-air
else {
        AttachAt(TraceHit.Location);
        AttachAt(TraceEnd);
}
```

Originally, I planned to have a complex inventory system, where players could hold multiple small items at once or have to use both their hands together to carry a larger one, making players have to think before picking up items willy-nilly. However, this came at a cost of having a more complex control system, which I did not find enjoyable. Having to switch modes to place items rather than using the same mode did not flow correctly with the game. I may come back to this system and implement it in a different way, but this may also require a full re-develop of the control scheme.

22 Hologram Bugfix

Commit

I finally worked out and fixed an error with the placer hologram, where placing and item, picking it up immediately and holding to place would not show the hologram due to a timer issue. This also allowed me to shorten the time required to hold to show the hologram.

23 Bin and Container Filter

Commit

I added a filter to the bin and the containers. This stops the player from being able to delete or store important items, such as chopping boards or knives.

```
void AParentBin::OnBRBeginOverlap...
// if AParentItem, destroy item
if (OtherActor->IsA(AParentItem::StaticClass())) {
        OtherActor->Destroy();
        AParentItem* ItemToBin = Cast<AParentItem>(OtherActor);
        if (ItemToBin->Data.bBinnable)
        {
                OtherActor->Destroy();
        }
}

bool AParentContainer::AddItemToContainer...
// Check if the item can be stored
if (ItemToAdd->Data.bStoreable) {
// Check if the container has an item stored already.
     //If it doesn't check if the new item ID can be stored
if (ContainedItem.ID == "" || Amount == 0) {
        if (AcceptedItems.Contains(ItemToAdd->Data.ID)) {
                // If the ID can be stored, set it to be this
↪  containers item
                SetupContainer(ItemToAdd->Data, 1);
                return true;
        }
}
```

24 Updating Containers

Commit

I removed the ContainerRange BoxCollision from the container, as it was unused. I also fixed a bug where when an item was placed in a container, the item held in the hand was destroyed but the HeldItem pointer was still stored. This stopped the player from being able to pick up any other item, interact with the container and would crash the editor when trying to place. When storing the pointer now becomes nullptr as well, after destruction. Additionally, when an item is removed from the container, if it would leave the container empty the stored item data is cleared.

Video

25 Trace Switch and Sink Start

Commit

The amount of results that could come from a trace was getting quite large, so I implemented a function, an enum and a switch to simplify it. ETraceQuery returns a value based on what type of item is hit, be it a basic item or a container, where is then is used in a switch on PrimaryActionPress, reducing the amount of if else statements used.

```
UENUM()
enum ETraceQuery
{
        HitActor UMETA(Display Name = "Hit Actor"),
        HitItem UMETA(Display Name = "Hit Item"),
        HitChopping UMETA(Display Name = "Hit Chopping Board"),
        HitPlate UMETA(Display Name = "Hit Plate"),
        HitContainer UMETA(Display Name = "Hit Container"),
        HitSink UMETA(Display Name = "Hit Sink"),
};

ETraceQuery APlayerCharacter::GetTraceHitClass(AActor*
↪  TraceOutput)
{
        // Query what a trace hit, returning an enum
        if (TraceHit.Actor->IsA(AParentItem::StaticClass())) {
↪  return ETraceQuery::HitItem; }
        else if
↪  (TraceHit.Actor->IsA(AParentStation::StaticClass())) {
↪  return ETraceQuery::HitChopping; }
        else if (TraceHit.Actor->IsA(APlate::StaticClass())) {
↪  return ETraceQuery::HitPlate; }
        else if
↪  (TraceHit.Actor->IsA(AParentContainer::StaticClass())) {
↪  return ETraceQuery::HitContainer; }
        else if
↪  (TraceHit.Actor->IsA(AParentSink::StaticClass())) { return
↪  ETraceQuery::HitSink; }
        return ETraceQuery::HitActor;
}
```

I also started on the ParentSink class. The sink has a simple role - washing plates when they become dirty. Plates can be added or removed from the sink in a similar way to adding an item to a container and interacting with the sink cleans the first dirty plate available.

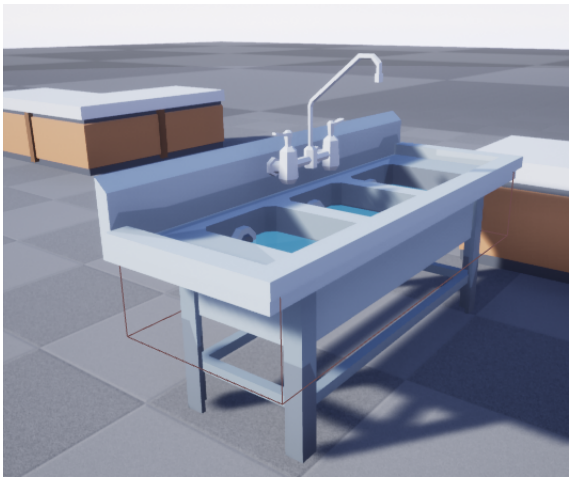

Figure 8. Sinks will mainly be used to clean plates

26 Cleaning Plates and GetTraceHitClass bugfix

Commit

I implemented the CleanPlate function next. When at least one dirty plate is in the sink interacting with the sink will clean it, simple enough. I also moved away from the container like system to a chopping board like system, where the player will just have to put the plate in the sink instead of interacting with it.

Finally, I refactored the order of GetTraceHitClass, changing from Item/Chopping/Plate/Container/Sink to Chopping/Plate/Container/Sink/Item, as Plates/Containers/Chopping are child classes of Item, they would return item first, rather than continue searching for their super classes.

27 Icon Creation Tool

Commit

I created a tool that allows me to create icons from the 3D models provided in the Synty packs that I am using. These icons are then used in multiple areas, such as the UI and on certain containers to show the player what is in them.

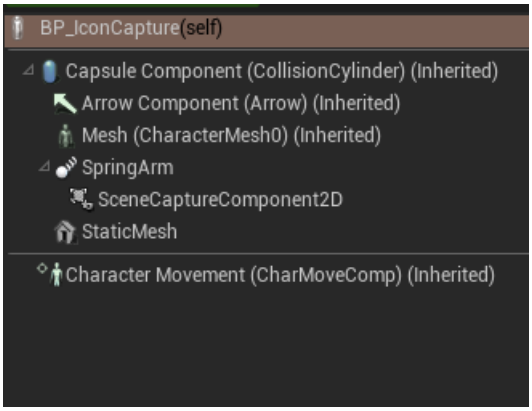

Figure 9. The component list of the Icon Tool

The tool is simply a spring arm attached to a static mesh component. A scene capture component is used to capture the image, set to an aspect ratio of 1.1 so the output image is a square. Additionally, any other objects other than the tool itself are culled from the rendered image, so only the static mesh is shown. From that render target, a separate texture is created through 'Create Static Texture', which is then used in a material.

For containers, the event 'SetContextData' is used, which is called whenever the item data is updated. The cardboard box simply takes the icon image material and sets two plane meshes to use the material.

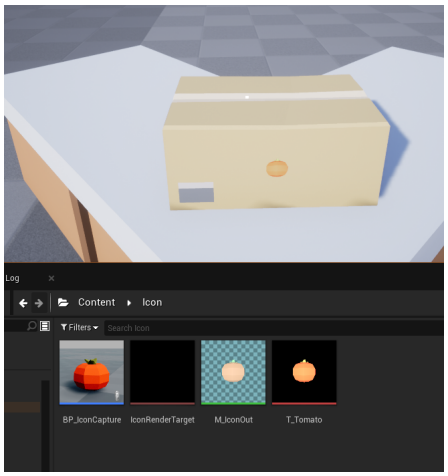

Figure 10. The box container using the icon. Other containers can use the 'SetContextData' function in different ways.

28 Tutorial Level

Commit

I took away from the development side to work on areas of design, starting with the tutorial level. Representing a 'game within a game', the tutorial level is like a computer generated restaurant for new managers to learn in before the real deal, so the world is contained in a simple white skybox. The area of play is small, being relegated to only the restaurant itself, but will serve its use as the learning environment well.

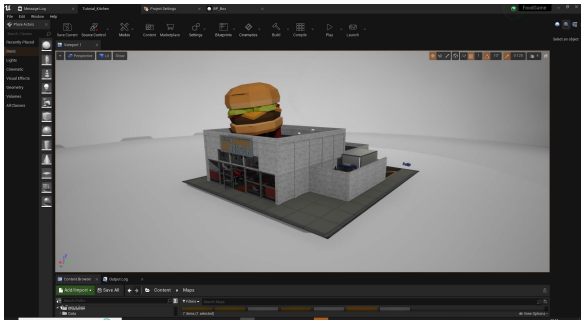

Figure 11. The tutorial level from a sky view. The white skybox emphasised that it is in a simulation rather than a world

Figure 11. The tutorial level from a front view.

During development, I would often oversize areas of the shop - such as the customer area and especially the kitchen. This would lead to long walk times between objectives when tested, further restricting the time limit set by the game. However I didn't want the areas to feel too small and cramped at the same time. I think the tutorial kitchen provides a good compromise.

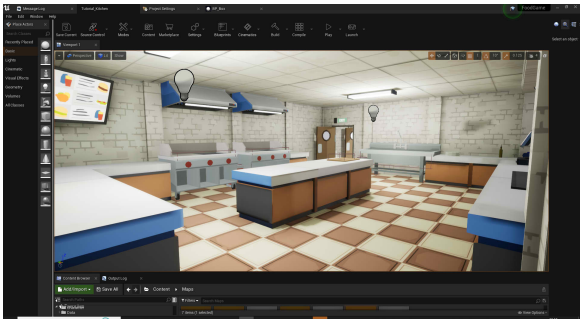

Figure 13. The tutorial level's kitchen. All storage was moved into a small shelving unit in the corridor

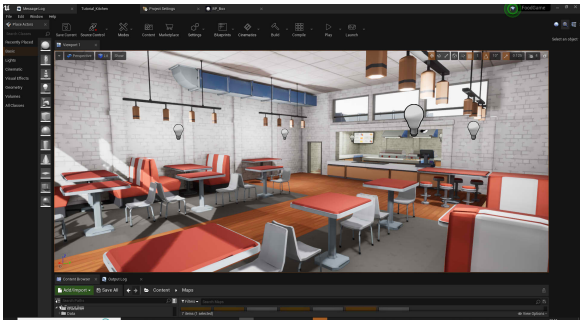

Figure 14. The tutorial level's dining area. Booths were used alongside tables to convey a greater use of the space

The back area of the restaurant features a slimmed down and closed off delivery area, with a path straight to the kitchen.

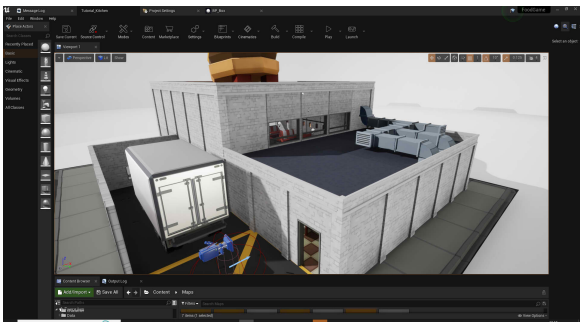

Figure 15. The tutorial level's delivery area. Closing off the delivery area tied it more too the restaurant rather than a parking space

I think the design and layout of the tutorial level is it's greatest strength. For an area to teach the player how to play the game, a smaller level is better. It also looks like a cafe or fast food joint rather than a high end restaurant. There are rooms for improvement, but this is mainly in the facade of it in a simulation. The kitchen provides a good compromise between oversized and cramped, with the central island allows the player to pass multiple items between the back wall and gantry. Changing the skybox from a smooth curve to a more boxy and polygonal one, while also changing the texture would provide a huge improvement.

29 Main Menu Character

Commit

Usage

2 Adding Chopping Recipes

To add a new chopping recipe, you simply open the Data.DT-ChoppingRecipes and add a new row to the table. The row name must be the same as the recipe ID and any items used in the row must have an string matching an item in the DT-Items data table.

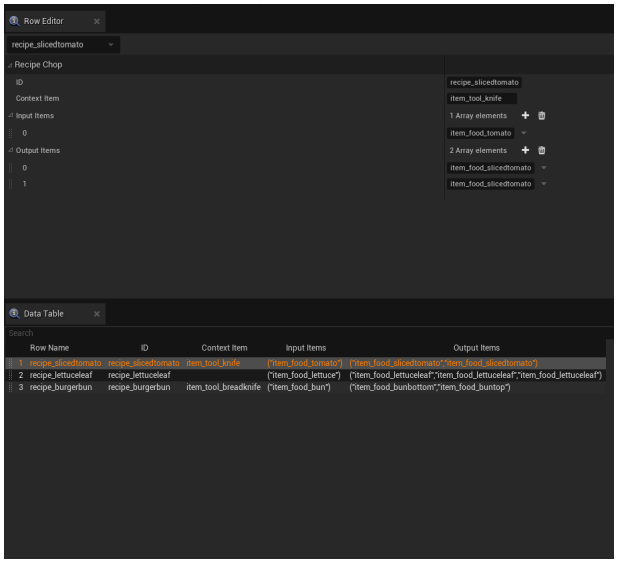

Figure 4. The Chopping Data Table, where the designer can easily implement a new chopping recipe

If a context item - such as a knife or utensil is needed - add that context item's ID to the Context Item Slot property. If no context item is needed, leave this blank.