CMP302 – Coursework Report

1800997 – Stylianos Zachariou

Summary

The game mechanic created, is a decapitation mechanic that allows the user to detach their head and arms and use them to move through tight spaces and possess other characters with different abilities. This drives inspiration from the game "Super Mario Odyssey", (Nintendo, 2017) where the player can possess other character using their hat, and from the game "Knack", (Japan-Studio, 2013) where the player can detach and control objects with parts of their body. The game mechanic consists of the following components:

- Three body parts that are detachable and moveable:
  - 'Head': needed to control all characters.
  - 'Left Arm' & 'Right Arm': needed to push objects.
- Three breakable characters, each with different abilities:
  - 'Plain Character': The starter character possessing no special abilities, but has their arms and head attached.
  - 'Jumping Character': A character that comes with no head or arms and can be controlled using only the head. Their ability is jumping higher than all other characters.
  - 'Strong Character': A character that comes with no head or arms. If both arms and the head are attached, this character can move heavy objects in the scene.
- Moveable objects that can only be moved when pushed by the 'Strong Character'
- 'Return to original body' system: transports body parts to their original body.
- Widgets showing if a body part can be attached to a character.

Demonstration Video Link:

https://youtu.be/ZjtVlYu5TTY

Requirements and Specifications

Introduction

Although a very small number of games include a detachable body part game mechanic, it is an interesting one full of potential. In this project an initial implementation of the game mechanic was created, exploring how its fundamental components could be constructed, keeping in mind future modification, extension, and improvement.

Furthermore, the game mechanic can be added to any game, and used to create intriguing levels for puzzle-platformers. The framework includes three controllable body parts, three characters with different abilities and moveable objects.

Perspective

The intention for this project was to create a detachable body part mechanic, where the player can detach their head and arms and control them to move around the level. The body parts can be re-attached to the original character or different characters offering different abilities.

Functions

- The user can detach their head and arms from their body.
- The user can control each body part separately, to move around the level.
- The user can return and attach any body part to its original body.
- The user can see when a body part can be attached to a character using UI.
- The user can control all characters by attaching the head on them.
- The user can push moveable objects when controlling the 'Strong Character' and both arms and head are attached.
- A programmer or designer can create new characters with different abilities.
- A programmer or designer can control the following without coding:
    - The movement speed of body parts.
    - The force that body parts have when detached.
    - The speed of body parts when returning to their original body.
    - The teleporting height of body parts while returning to their original body.
    - The speed of body parts when attaching to a new body.
    - The jump height of the jumping character.

Design and Implementation

The project is directed to be used by a game designer or software engineer who is acquainted with the Unreal Engine 4. For small modifications on the existing implemented material (e.g., Changing meshes or speed and movement variables) no programming knowledge is needed. However, to extend or improve the current framework (e.g., creating new characters with different abilities), knowledge of the C++ programming language is required.

System Features

Body Part: Head

Description:
A body part that is originally attached to the plain player. Once detached, the head can be rolled around the level to enter small spaces and possess other characters with special abilities.

Stimulus / Responsible sequences:
For gameplay, the head can be controlled using the WASD keys.

For development, the head movement variables can be controlled in the UE4 blueprint editor of the character that spawned it. Otherwise, the head C++ code can be opened in Visual Studio.

Functional requirements:

- For the head to move, it must be detached and not connected to any character and be selected using the keyboard button used to originally eject it.
- The head can be detached, with a specific force, from the body is currently attached to.
- The head can return to its original body using the "R" keyboard button.
- The head can be attached to any character.

Body Part: Right Arm

Description:
A body part that is originally attached to the plain player. Once detached, the right arm can be rolled around the level to enter small spaces. Unlike the head, the right arm can only connect to the 'Strong Character' and 'Plain Character'.

Stimulus / Responsible sequences:
For gameplay, the right arm can be controlled using the WASD keys.

For development, the right arm movement variables can be controlled in the UE4 blueprint editor of the character that spawned it. Otherwise, the right arm C++ code can be opened in Visual Studio.

Functional requirements:

- For the right arm to move, it must be detached and not connected to any character and be selected using the keyboard button used to originally eject it.
- The right arm can be detached, with a specific force, from the body is currently attached to.
- The right arm can return to its original body using the "R" keyboard button.
- The right arm can be attached only to the 'Plain' and 'Strong' characters.

Body Part: Left Arm

Description:
A body part that is originally attached to the plain player. Once detached, the left arm can be rolled around the level to enter small spaces. Unlike the head, the left arm can only connect to the 'Strong Character' and 'Plain Character'.

Stimulus / Responsible sequences:
For gameplay, the left arm can be controlled using the WASD keys.

For development, the left arm movement variables can be controlled in the UE4 blueprint editor of the character that spawned it. Otherwise, the left arm C++ code can be opened in Visual Studio.

Functional requirements:

- For the left arm to move, it must be detached and not connected to any character and be selected using the keyboard button used to originally eject it.
- The left arm can be detached, with a specific force, from the body is currently attached to.
- The left arm can return to its original body using the "R" keyboard button.
- The left arm can be attached only to the 'Plain' and 'Strong' characters.

Character: Plain Character

Description:
The plain character is the starter character used to spawn the body parts.

Stimulus / Responsible sequences:
For gameplay, the plain character can be controlled using the WASD keys and jump using SPACEBAR.

For development, the plain character blueprint can be opened in the UE4 blueprint editor, and its C++ code can be opened in Visual Studio

Functional requirements:

- For the plain character to move the head needs to be attached to it.
- The plain character has a medium jump.

Character: Jumping Character

Description:
The jumping character is a special ability character that can jump higher than any other character.

Stimulus / Responsible sequences:
For gameplay, the jumping character can be controlled using the WASD keys and jump using SPACEBAR.

For development, the jumping character blueprint can be opened in the UE4 blueprint editor, and its C++ code can be opened in Visual Studio

Functional requirements:

- For the jumping character to move the head needs to be attached to it.
- The jumping character has a very high jump.

Character: Strong Character

Description:
The strong character is a special ability character that can push heavy objects in the level.

Stimulus / Responsible sequences:
For gameplay, the strong character can be controlled using the WASD keys and jump using SPACEBAR.

For development, the strong character blueprint can be opened in the UE4 blueprint editor, and its C++ code can be opened in Visual Studio

Functional requirements:

- For the strong character to move the head needs to be attached to it.
- For the strong character to push heavy objects, both arms and the head must be attached to it.

- The strong character has a very low jump.
- The strong character has a different animation and walking speed when pushing a moveable object.
- When detaching any body parts from the strong character, all body parts should come off.

Moveable Object

Description:
The moveable object is a heavy object that can be placed in the level and can only be moved by the strong character.

Stimulus / Responsible sequences:
For gameplay, the moveable object can not be controlled directly by the user.

For development, the moveable object blueprint can be opened in the UE4 blueprint editor, and its C++ code can be opened in Visual Studio

Functional requirements:

- For the moveable object to move, the user must control the strong character with all body parts attached and push it towards the desired direction.

Connect Widget

Description:
Each character has widgets, made visible only when a body part that can be connected is in reach.

Stimulus / Responsible sequences:
For gameplay, the widget cannot be controlled directly by the user.

For development, the widget blueprint can be opened in the UE4 blueprint editor.

Functional requirements:

- For the widget object to be visible, a body part that can connect to the player must be in range.
- The widget must disappear when the body part connects to the character.
- The widget must be in the same position as the character it is attached to.

Methods

Body part detachment

An essential requirement of the game mechanic is detaching the body parts from the player's body. To achieve this, each body part is bound to a key bind and can be detached using the keyboard keys 1-3. In the Unreal Editor, the input actions are set up as 'HeadOff', 'RightArmOff' and 'LeftArmOff'. When any of these keys are pressed the respective body part is detached, thrown with a customizable force away from the body and automatically possessed by the player. The body part's collisions and physics calculations are also enabled, allowing it to roll on the ground instead of going through it. The force applied when detaching a body part can be changed in the UE4 Blueprint editor of the character. (Figure 1)

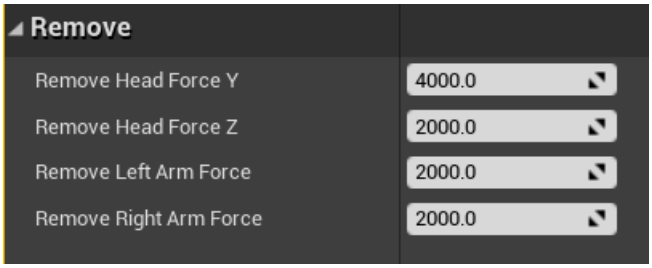


Figure 1- Force Variables Editable in the UE4 Blueprint Editor

Body part selection

When all body parts are detached, it is required for the player to be able to switch between them and control them. If a body part is detached the player can press the same button used to detach it to control it. This is done by using the same key bindings used for detachment and checking if the current body part is attached to the body. If it's not attached, then it is possessed by the player.

Body part return

If a body part is detached, it can be returned to its original body automatically by only using a key press. This is done by using a key bind for the return which is the "R" keyboard button. In the Unreal Editor, the input action is set up as 'Return'. When a body part is not connected to any character and the return action is called, the body part flies into the air with a spinning animation. When it reaches a certain height, the body part teleports to the x and y coordinates of the original body and then starts moving downwards towards its z coordinate. Once it reaches the position, the body part rotates until its rotation is the same as the characters. When both the location and rotation are the same as the original characters, the body part is attached, and the player possesses the original character. (Figure 3)

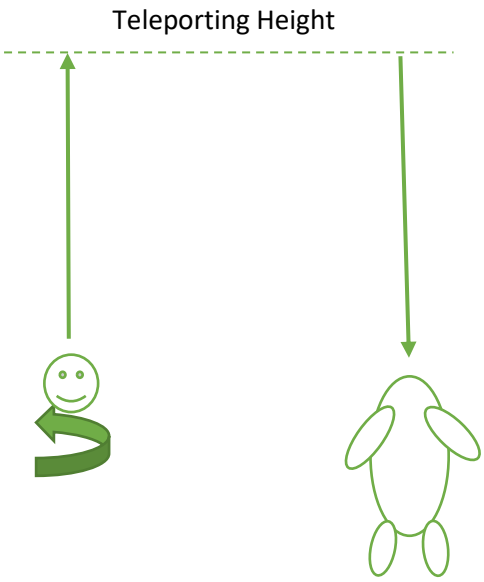


Figure 3- Diagram of Head Motion When Returning to its Original Body

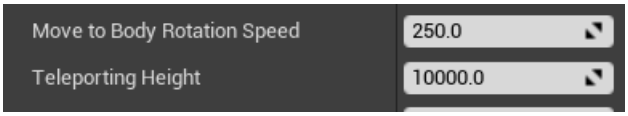


Figure 2-Editable Variables in the UE4 Blueprint Editor About Returning to Original Body

Body part connect to character

When a body part is near a character, it can be attached to it by using the same keyboard button used to detach the specific body part. As soon as the button is pressed, the body part starts rotating and moving until it matches the same rotation and location as the character. If the body part attached is the head, the player then possesses the newly connected character. However, if the body part connected is one of the arms, then the player possesses the original body again, this is done so that only the character connected to the head can be controlled. The speed of the body part connecting to the character can be changed in the UE4 Blueprint Editor. (Figure 4)

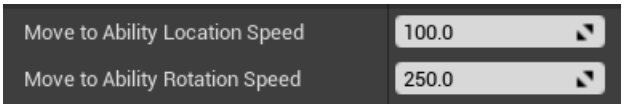


Figure 4-Editable Variables in the UE4 Blueprint Editor About Connecting to Ability Character

Body part movement

As soon as a body part is detached, it can move freely in the level using the WASD keys. A force is applied to the body part when any of these keys are pressed making it move towards the desired direction. Some linear damping is also added to the body part to make it easier to control and impossible to move uncontrollably fast. The movement speed can be modified in the UE4 Blueprint editor. (Figure 5)

Figure 5-Editable Variables in the UE4 Blueprint Editor About Body Part Movement

Jumping Character jump

The jumping character can jump higher than all other characters in the scene and is possessed by the player when the head is attached to them. The player can make the character jump using the

SPACEBAR bound to the action ‘Jump’ in the Unreal Editor. This character can jump higher because its jump z velocity is much higher than the other characters. The jump z velocity can be modified easily in the blueprint editor, under the movement character component. (Figure 6)

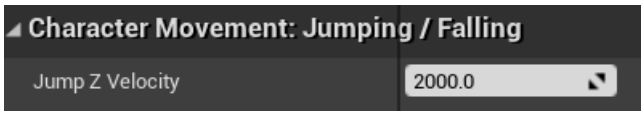


Figure 6- Jump Velocity Variable Editable from the UE4 Blueprint Editor

Strong Character pushing and moveable objects

The strong character can move heavy objects in the scene by pushing them. To do that, both arms and head must be attached to it. Once the character starts pushing on the moveable object, the object detects the player using a collision box and checks if all body parts are attached to it. If the character is complete, the moveable object’s, mass is lowered, making the strong character able to push the object in any desired direction. While pushing the strong characters animation is also changed to a pushing animation, and its speed is lowered creating the illusion that the object is heavy.

Character Widgets

Each character has a widget for every body part that can connect to it. The jumping character can only have the head connected to it; therefore, one widget is used. On the other hand, the strong character needs three widgets since all body parts can connect to it. When a body part moves near a character, a collision sphere around the character detects it and depending on the body parts type, the respective widget is made visible. The widgets are used to signal to the player when and how a body part can connect to a character. (Figure 7)

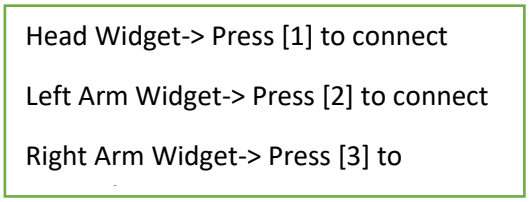
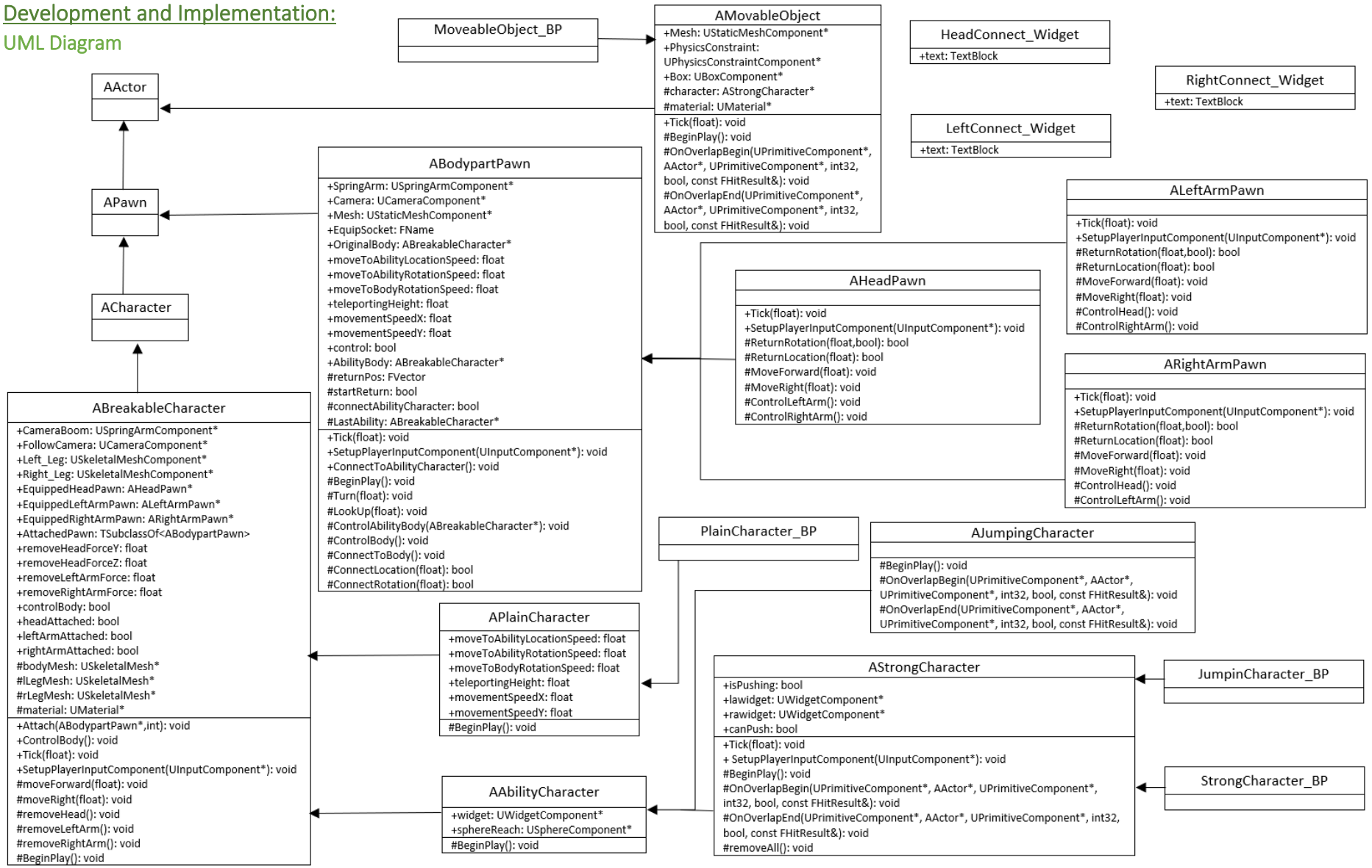


Figure 7-Character Widgets for Different Body Parts

Development and Implementation:

UML Diagram



Technical Discussion:

Body parts:

To create each of the body parts, a general “BodypartPawn” class was created, initializing variables and functions, essential for each body part to behave properly. Components such as a spring arm, a camera and a static mesh component are created so that the user can follow the body part using the camera and view the body part with the mesh. Each body part also needs the socket name of the socket that it will connect to on the character’s body’s skeletal mesh and a pointer to its original body; useful for returning to the original body. A Boolean of whether the user is controlling the body part is also needed, so that it doesn’t try to move if it’s not possessed. Variables that allow easy modification from the UE4 Blueprint Editor are also initialized along with two Boolean variables triggered, when the body part must return to its original body or when the body part must connect to an ability character.
In the constructor, the spring arm is made to rotate independently from the mesh, to allow the camera to remain steady while the body part is rolling around. A camera is then created, and the Mesh is made to have collisions enabled. In the “BeginPlay” function, the mesh is set to simulate physics, allowing the addition of forces when the body part must move.
In the “SetupPlayerInputComponent”, two functions (“Turn” and “LookUp”) are bound to axis “Turn” and “LookUp” respectively and the “ControlBody” function is bound to the “ControlBody” action (triggered when the R key is pressed). The “Turn” function adds to the controller’s yaw input, affecting the movement of the pawn, and then gets the controller’s yaw rotation, applying it to the Spring Arm’s relative rotation. (Figure 8) The “LookUp” function has the same behaviour but instead of affecting the Yaw rotation, it affects the Pitch rotation.

```
void ABodypartPawn::Turn(float AxisValue)
{
    if (control)
    {
        AddControllerYawInput(AxisValue);
        FRotator Rotation = Controller->GetControlRotation();
        SpringArm->SetRelativeRotation({ SpringArm->GetRelativeRotation().Pitch, Rotation.Yaw, 0 });
    }
}
```

Figure 8-The "Turn" function

The “ControlBody” function is responsible for unpossessing the body part and possessing the original body. Also, it makes the control variable’s value to false, so that the pawn doesn’t try to move while not being possessed. A similar function of the class is the “ControlAbilityBody” function which takes a breakable character and performs the same operations to possess that character. (Figure 9)

```
void ABodypartPawn::ControlAbilityBody(ABreakableCharacter* body)
{
    APlayerController* controller = GetWorld()->GetFirstPlayerController();
    controller->UnPossess();
    controller->Possess(body);
    body->ControlBody();
    control = false;

}
```

Figure 9-The "ControlAbilityBody" function

The last two functions defined in the class are “ConnectLocation” and “ConnectRotation” which are both used to move the body part from its current position and rotation to the ability body’s location and rotation. In the “ConnectRotation” function the rotation of the socket that the body part will

connect is fetched and using the “moveAbilityRotationSpeed” variable the body part slowly rotates to that rotation. The same principle is followed by the “ConnectLocation” function, but instead of the Pitch, Yaw and Roll, the xyz coordinates are compared. (Figure 10)

```
if (GetActorLocation().X < bodyLocation.X - moveAmount)
{
    SetActorLocation({ GetActorLocation().X + moveAmount, GetActorLocation().Y,GetActorLocation().Z });
}
else if (GetActorLocation().X > bodyLocation.X + moveAmount)
{
    SetActorLocation({ GetActorLocation().X - moveAmount,GetActorLocation().Y,GetActorLocation().Z });
}
else
{
    SetActorLocation({ bodyLocation.X,GetActorLocation().Y,GetActorLocation().Z });
}
```

Figure 10-The x-coordinate calculation from the "Connect Location" function

The last main function created in this class is the “ConnectToAbilityCharacter” function. It checks whether there is an ability character nearby and connects to it by turning on the “connectAbilityCharacter” Boolean and disable the body part’s collisions. A pointer to the ability character is stored along with a vector of the body part’s current position, used to connect the body part to the ability character’s position.

One of the body parts inheriting from the body part class is the “HeadPawn” class. In its constructor, the mesh is set and the equip socket’s name defined as “headSocket”. Then, this class extends the “BodyPart” class by firstly adding the movement functions “MoveForward” and “MoveRight” for the head. These two functions are bound to Axis “MoveForward” and “MoveRight” respectively and are responsible for adding a force to the player in the desired direction. The “MoveForward” function gets the Yaw rotation of the controller and uses it to get the unit axis X. It uses that axis to get a direction and using that direction, the movement speed is applied using the “AddForce” function. (Figure 11) The movement speed of the head is always half the one of the arms since its shape is spherical it needs less force to move.

```
void AHeadPawn::MoveForward(float AxisValue)
{
    if (control)
    {
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation(0, Rotation.Yaw + 90, 0);

        const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);

        Mesh->AddForce({ (Direction.X) * AxisValue * movementSpeedX/2 ,(Direction.Y) * AxisValue * movementSpeedY/2 ,0 });
    }
}
```

Figure 11- The "MoveForward" function

The Move Right function uses the same calculations but to get the direction, instead of using the x-axis, it uses the y-axis. As the other two body parts classes, this class has functions to possess any of the other body parts of the original body. This is done by getting the body part that we want to control from the pointer that already exists to the original body. Doing that the other body part can easily be possessed using the same functions used for the “ControlAbilityBody” class.

Returning to its original body is something essential for any body part, therefore, the “ReturnRotation” and “ReturnLocation” functions are created responsible for returning the head smoothly back to its body. The aim of the two functions is to make the head fly into the air while continuously rotating on the y-axis. When the head reaches the teleporting height, it must teleport to the xy coordinates of the socket that it will connect to and thereafter, move downwards until it reaches the correct z-coordinate. When the correct location is reached, the head continues to rotate until it reaches the

rotation of the socket; creating an illusion that the head is being screwed back into its place. To achieve this, firstly, the highest point, the head must reach, must be calculated by finding whether the head or the body is in a higher position and adding the “teleportingHeight” variable to the highest location. This is done so that even if the body is in a much higher position than the head the head will travel high enough to surpass that position. The movement speed is also calculated by subtracting the lowest point from the highest point, so that if the highest point and the lowest point are very far apart, the head will move faster, so that the user doesn’t have to wait for long. Then, if the head position is lower than the highest point, the movement speed multiplied by the Delta Time is added to the location. If the highest point is reached, the heads x and y coordinates become the socket’s x-y coordinates (Figure 12)

```
if (GetActorLocation().Z < highestPoint)
{
    SetActorLocation({ GetActorLocation().X, GetActorLocation().Y, GetActorLocation().Z + movementSpeed * DeltaTime });
}
else
{
    SetActorLocation({ bodyLocation.X, bodyLocation.Y,  GetActorLocation().Z + movementSpeed * DeltaTime });
}
```

Figure 12-Moving the head to the highest point and then teleporting it.

Then if the head’s xy coordinates are the same as the socket’s the head moves down with the same speed until it reaches the socket’s z-coordinate, and the function then returns true. Meanwhile the “ReturnRotation” function checks if the “ReturnLocation” function returned true. If it returned true the function matches the head’s Yaw rotation to the body’s Yaw rotation, else it matches the pitch and roll but keep incrementing the Yaw rotation to create the spinning effect. (Figure 13)

```
if (arrived)
{
    if (GetActorRotation().Yaw < bodyRotation.Yaw - rotateAmount)
    {
        SetActorRotation({ GetActorRotation().Pitch ,GetActorRotation().Yaw + rotateAmount,GetActorRotation().Roll });
    }
    else if (GetActorRotation().Yaw > bodyRotation.Yaw + rotateAmount)
    {
        SetActorRotation({ GetActorRotation().Pitch ,GetActorRotation().Yaw + rotateAmount,GetActorRotation().Roll });
    }
    else
    {
        SetActorRotation({ GetActorRotation().Pitch,bodyRotation.Yaw,GetActorRotation().Roll });
    }
}
else
{
    SetActorRotation({ GetActorRotation().Pitch ,GetActorRotation().Yaw + rotateAmount,GetActorRotation().Roll });
}
```

Figure 13-Heads Yaw Rotation Calculation in the "ReturnRotation" function

Lastly, the “HeadPawn” class binds the “ConnectToAbilityCharacter” function to the “HeadOff” action, so that the head can connect to ability characters using the same button that detaches it.
Similarly, the “LeftArmPawn” and “RightArmPawn” classes include the same functions, but they are implemented in a slightly different way. Firstly, in their constructors the meshes are set and the names of the equipped sockets are defined (“leftArmSocket” and “rightArmSocket”). The moving functions although very similar, instead of half the movement speed, they apply the full movement speed variable since the arms need more force to move. The “ReturnLocation” function is the same, however the “ReturnRotation” function, again continuously rotates the Yaw until the body part reaches its desired location but this time waits to adjust the pitch and roll rotations until the body part arrives at the socket’s location, creating a robotic assembly effect. Lastly, the “ConnectToAbilityCharacter” function is bound to the “LeftArmOff” action in the “LeftArmPawn” class and on the “RightArmOff” action in the “RightArmPawn” class.

Breakable Characters:

The “BreakableCharacter” class which inherits from the “Character” class is inherited by all playable characters. It includes essential components like a spring arm component, camera component and two skeletal mesh components (one for each leg). Three pointers are also initialized; one for each body part and a class reference of the body part class is initialized, which will later help when attaching body parts to the character. Four variables are defined to offer customizability in the blueprint editor for the force applied to each body part when detaching from the body. Three booleans, each portraying the attachment status of a body part are also initialised along with a “controlBody” boolean which is true only when the player controls the character. The skeletal meshes and the material of the three unattachable body parts (torso and legs) are also initialized and stored in this class.
The functions created in this class are identical across all characters, therefore, they are initialized and defined in this general class. Firstly, the movement functions “moveForward” and “moveRight” are the same as the homonymous functions in the body part classes and have the same functionality. The “ControlBody” function is called when the character is going to be controlled by the player. It un-pauses, the players animation and makes the control boolean true if the head is attached to the character. The “removeHead”, “removeLeftArm” and “removeRightArm” are all functions responsible for detaching each of the body parts from the character. All three are identical in their operations but each operates on a different body part. The functions are only carried out if the respective body part is attached to the body. The control Boolean of the character becomes false, and the control Boolean of the body part becomes true. The body part is detached using the “DetachFromComponent” function and set to simulate physics and collisions. The location and rotation of the body part’s socket is fetched and applied to it so that it will seamlessly detach. Then using a rotation matrix, an impulse is applied to the body part, so that it will pop of the body with some force. (Figure 14) Furthermore, the character is unpossessed, and the body part is possessed. The character’s animations are also reset and paused.

```
//Force
FVector imp = { 0,removeHeadForceY,removeHeadForceZ };
FRotator rot = GetMesh()->GetComponentRotation();
FRotationMatrix matrix = FRotationMatrix(rot);
imp = UKismetMathLibrary::Matrix_TransformVector(matrix, imp);
EquippedHeadPawn->Mesh->AddImpulse(imp);
```

Figure 14-Adding the impulse to the head in the "removeHead" function

These three functions are also used when each body part is not attached to the body, to switch control between the character and the respective body part. (Figure 15)

```
APlayerController* controller = GetWorld()->GetFirstPlayerController();
controller->UnPossess();
controller->Possess(EquippedHeadPawn);
EquippedHeadPawn->control = true;
controlBody = false;

GetMesh()->InitAnim(true);
GetMesh()->bPauseAnims = true;
Left_leg->InitAnim(true);
Left_leg->bPauseAnims = true;
Right_leg->InitAnim(true);
Right_leg->bPauseAnims = true;
```

Figure 15-Possessing the head code in the "removeHead" function

The “Attach” function is used to attach any body part to the character. Depending on what body part is getting attached, this function firstly makes the equivalent body part pointer to point to it. It also uses the “AttachToComponent” function to attach the body part to the required socket. Then the attached Boolean of the body part is turned to true. If the body part connected is the head, then the “ControlBody” function is called, so that the character is possessed only if the head is attached to it. (Figure 16)

```
if (bodypart == 1)
{
    EquippedHeadPawn = (AHeadPawn*)newAttachment;
    EquippedHeadPawn->AttachToComponent(GetMesh(), rules, EquippedHeadPawn->EquipSocket);
    headAttached = true;
    ControlBody();
}
```

Figure 16-Attaching the head using the "Attach" function

Bringing all these together, in the “SetupPlayerInputComponent” function, the “moveForward” and “moveRight” functions are bound to Axis “MoveForward” and “MoveRight” while the “Jump” and “StopJumping” functions that are inherited by the “Character” class are bound to action “Jump”. To make the camera turn, the “AddControllerYawInput” and “AddControllerPitchInput” functions, inherited by the “Pawn” class are bound to Axis “Turn” and “LookUp”. The three functions “removeHead”, “removeLeftArm” and “removeRightArm” are bound to actions “HeadOff”, “LeftArmOff” and “RightArmOff” respectively. Lastly, in this class’s constructor, the camera is set up and connected to the spring arm. The three static body part’s (torso and legs) meshes are created, saved, and later set in the “BeginPlay” function.

The first class inheriting from the “BreakableCharacter” class is the “PlainCharacter” class. The plain character class is a class responsible of extending the simple breakable character into a character that spawns in with all body parts attached. This class includes all the customizable variables that affect the body parts so that they can be easily accessed through the plain character’s blueprint in the UE4 Blueprint editor. In its constructor, this class finds and stores the material used to dress the meshes. Then, the begin function is called which spawns and attaches all three body parts to the character. To do this, each body part is spawned using the “SpawnActor” function, its collisions and physics are turned off and the customizable variables are applied to it. Moreover, it is attached to the body using the “AttachComponent” function and the corresponding socket. Lastly, for each body part, the original body pointer is made to point to the plain character. (Figure 17)

```
AttachedPawn = AHeadPawn::StaticClass();
EquippedHeadPawn = GetWorld()->SpawnActor<AHeadPawn>(AttachedPawn, transform.GetLocation(), transform.GetRotation().Rotator(), spawnParams);
EquippedHeadPawn->SetActorEnableCollision(false);
EquippedHeadPawn->DisableComponentsSimulatePhysics();
EquippedHeadPawn->moveToAbilityLocationSpeed = moveToAbilityLocationSpeed;
EquippedHeadPawn->moveToAbilityRotationSpeed = moveToAbilityRotationSpeed;
EquippedHeadPawn->moveToBodyRotationSpeed = moveToBodyRotationSpeed;
EquippedHeadPawn->teleportingHeight = teleportingHeight;
EquippedHeadPawn->movementSpeedX = movementSpeedX;
EquippedHeadPawn->movementSpeedY = movementSpeedY;

FAttachmentTransformRules rules(EAttachmentRule::SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepRelative, true);
EquippedHeadPawn->AttachToComponent(GetMesh(), rules, EquippedHeadPawn->EquipSocket);
EquippedHeadPawn->OriginalBody = this;
```

Figure 17-Spawn a Head in the world and attach it to the plain character

The “AbilityCharacter” class is the second class inheriting from the “BreakableCharacter” class. This is the base class for building characters with special abilities. It includes a widget and a sphere component. The widget is used to signal to the user when a body part can be connected to a character, whereas the sphere is used to detect any overlaps of body parts and make the widget visible.

Two classes inherit from the “AbilityCharacter” class: “JumpingCharacter” and “StrongCharacter”. The “JumpingCharacter” class is responsible for creating a character that can be controlled using the head and can jump very high. Only two new functions are needed: “OnOverlapBegin” which will be called wherever an overlap is detected by the sphere and “OnOverlapEnd” which will be called whenever the overlap ends. The overlap functions set the widget’s visibility to true when there is an overlap between the sphere and a head and false when the overlap ends. Whenever there is overlap, they also set the head’s “AbilityBody” pointer to point at the jumping character. When the overlap ends, the pointer is made to point to NULL. (Figure 18)

```
void AJumpingCharacter::OnOverlapBegin(class UPrimitiveComponent* OverlappedComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (OtherActor->GetClass() == AHeadPawn::StaticClass())
    {
        widget->SetVisibility(true);
        AHeadPawn* headPawn = (AHeadPawn*)OtherActor;
        headPawn->AbilityBody = this;
    }
}

void AJumpingCharacter::OnOverlapEnd(class UPrimitiveComponent* OverlappedComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
    if (OtherActor->GetClass() == AHeadPawn::StaticClass())
    {
        widget->SetVisibility(false);
        AHeadPawn* headPawn = (AHeadPawn*)OtherActor;
        headPawn->AbilityBody = NULL;
    }
}
```

Figure 18- The two functions called at the beginning and end of an overlap

In the constructor of the class the “JumpZVelocity” from the character’s movement component is incremented by a high amount, making the character able to jump very high. The material used for the static body parts is also found and stored. In the “BeginPlay” function the material of the body parts is set to the stored material and the two overlap functions are set as the sphere’s overlap functions.

The last character class is the “StrongCharacter” class. This class is responsible for creating a character that can move heavy objects which are unmoveable by other characters. To make the character able to push all three body parts must be able to connect to it. Therefore, two more widgets must be created (one for each arm). Two booleans are also initialized, one signalling when the character is pushing and one signalling whether the character can push. Also, two overlap functions must be created as for the “JumpCharacter” class and a “removeAll” function that removes all body parts from the character.

Firstly, in the constructor, the character’s “JumpZVelocity” is set to a low amount (making the character unable to jump high), the material for the static body parts is found and stored and the widgets are created. The “removeAll” function bound to all three remove keys is calling all three “removeLeftArm”, “removeRightArm” and “removeHead” functions so that all body parts are detached at once. The overlap functions are identical to the jumping character’s class, but they act on all three body parts, since all three can be attached to the character. Lastly, in the “Tick” function, the “canPush” Boolean becomes true only if both arms are attached, making the character unable to push if all body parts are not attached. If the body is controlled the sphere’s collision is turned off, so that the player can’t see the widgets while controlling the character. Finally, if the character can push and is pushing, the characters maximum walk speed is lowered; creating the illusion that the object being moved is heavy. (Figure 19)

```
if (canPush)
{
    if (isPushing)
    {
        GetCharacterMovement()->MaxWalkSpeed = 200;
    }
    else
    {
        GetCharacterMovement()->MaxWalkSpeed = 600;
    }
}
```

Figure 19-Character's speed being changed if they are pushing

Moveable Object:

The last class used for the game mechanic is the “MoveableObject” class. A moveable object is a heavy object that can only be moved by the strong character. Firstly, the class inherits from the “Actor” class and

extends it by adding a static mesh component, a physics constraint component, and a box component. A pointer to a strong character is also included along with two overlap event functions (begin and end). In the constructor, the mesh component is created, and the material used for the static body parts is found and stored. The box is set to detect collisions, generate overlap events, and have an "Overlap" collision response to all channels. The Physics constraint component is set to have the z-axis locked, so that it cannot be lifted, but the other two axis free to move. Also, the twist, swing 1 and swing 2 limits are set to locked so that the actor will not twist or turn. Furthermore, the constrained component is set to the Mesh. (Figure 20)

```
PhysicsConstraint->SetLinearZLimit(ELinearConstraintMotion::LCM_Locked,1);
PhysicsConstraint->SetLinearXLimit(ELinearConstraintMotion::LCM_Free, 1);
PhysicsConstraint->SetLinearYLimit(ELinearConstraintMotion::LCM_Free, 1);
PhysicsConstraint->SetAngularTwistLimit(EAngularConstraintMotion::ACM_Locked, 1);
PhysicsConstraint->SetAngularSwing1Limit(EAngularConstraintMotion::ACM_Locked, 1);
PhysicsConstraint->SetAngularSwing2Limit(EAngularConstraintMotion::ACM_Locked, 1);
PhysicsConstraint->SetConstrainedComponents(Mesh, FName("Mesh"),NULL,NAME_None);
```

Figure 20-Physics Constraint Set Up

The Mesh's mass is also set using the "SetMassOverrideInKg" to a very high amount, making it very hard to move by any player.
The "OnOverlapBegin" function checks whether the overlapped character is of the "StrongCharacter" class and if its true, the character pointer points to that strong character. On the other hand, the "OnOverlapEnd" function checks if the actor that ended overlap is the same as the character pointer. If this is true, the character's "isPushing" variable is turned to false and the character pointer points to NULL. In the "BeginPlay" function the Mesh is set to simulate physics and the mesh's material is set to the stored material. The overlap functions are also set as the box's overlap functions. Lastly, in the "Tick" function, if there is not a character nearby, the of the object is set to extremely high. If a character is in range, can push and their velocity is more than 0, the character's is pushing variable becomes true and the meshes mass is lowered, rendering the strong character able to push it.

Development Process:

Before development I created a brainstorm diagram to think of ideas to include in my mechanic. After choosing the best ideas, I created some general Pseudocode and then moved straight to C++. I decided to leave some parts (e.g., widgets and adding animation to characters) in blueprint since the implementation was much quicker and easier.

Conclusion:

To conclude, my game-mechanic came out better than expected. At the beginning I though it would be a very simple implementation, but as I was implementing more ideas, the code was becoming more and more complex. The controls sometimes become confusing, so I would like to add some UI where the player can choose what body part to control using their mouse, instead of the keyboard buttons. Although satisfied, if I had more time, I would have loved to polish the game-mechanic more, adding particle effects, more complex animations, and even more characters with more complex abilities. Overall, for the time spent on the project, I find the game mechanic in a very advanced level, and I can't wait to extend on it on my own time without a time limit.

References

Adobe, 2008. *Mixamo.* [Online]
Available at: https://www.mixamo.com/#/?page=1&type=Motion%2CMotionPack
[Accessed 17 January 2022].

Japan-Studio, 2013. *Knack.* s.l.:Sony Computer Entertainment.

Nintendo, E., 2017. *Super Mario Odyssey.* s.l.:Nintendo Entertainment Planning & Development.

Uisco, n.d. *UE4 Mannequin Dismemberment.* [Online]
Available at: https://uisco.itch.io/ue4-mannequin-dismemberment?download
[Accessed 2022 January 2022].