Appendix C: Technical Documentation

Name	Super Class	Description
ABody	AActor	Body Template Class.
ABodyModel	AUserRepresentation	Container class for Model user representation
ADPCharacter	APawn	Main class of user representation
ADPGameModeBase	AGameModeBase	Contains Main settings.
AFootball	AGoalGame	Game – Kicking ball into goal.
AForest	AActor	Tree grid 10 by 10 of randomly chosen trees with random rotation and location between 225 and 275 centimeters from each other.
AGame	AActor	Game template class.
AGoalGame	AGame	Common superclass of Football and hockey, contains common elements of both games.
AHand	AActor	Hand template Class.
AHockey	AGoalGame	Hockey Game – shooting puck into goal.
AHockeyStick	AActor	Hockey Stick.
AJointBody	ABody	Implements processing of hand data for joint user representation
AJointHand	AHand	Implements processing of body data for joint user representation
AJointSkeleton	AUserRepresentation	Container class for Joint user representation
AMenu	AActor	Menu (tablet PC with 4 options).
AModelBody	ABody	Implements processing of body data for model user representation
AModelHand	AHand	Implements processing of hand data for model user representation
ATowerOfHanoi	AGame	Game of Tower of Hanoi.
ATowerOfHanoiDisk	AActor	One disk takeable by user.
AUserRepresentation	AActor	Contains main logic of updating user-representation in scene.
AVehicle	AWheeledVehicle	Vehicle.
KinectBridge	N/A	Gets data from Kinect and converts into UE4 coordinates.
LeapBridge	N/A	Gets data from LM and converts into UE4 coordinates.
TakeRules	N/A	Rules for taking object.
UTakeHandler	USceneComponent	Handles hand (take) interaction.
UVehicleWheelFront	UVehicleWheel	Front wheel of vehicle.
UVehicleWheelRear	UVehicleWheel	Rear wheel of vehicle.

Table 1: Classes Overview (Gray super classes are provided by UE4)

Note that in UML Class diagrams; UE4 Base classes are shown in dark gray color and doesn't display complete UE4 (only overridden functions) class and inheritance hierarchy.

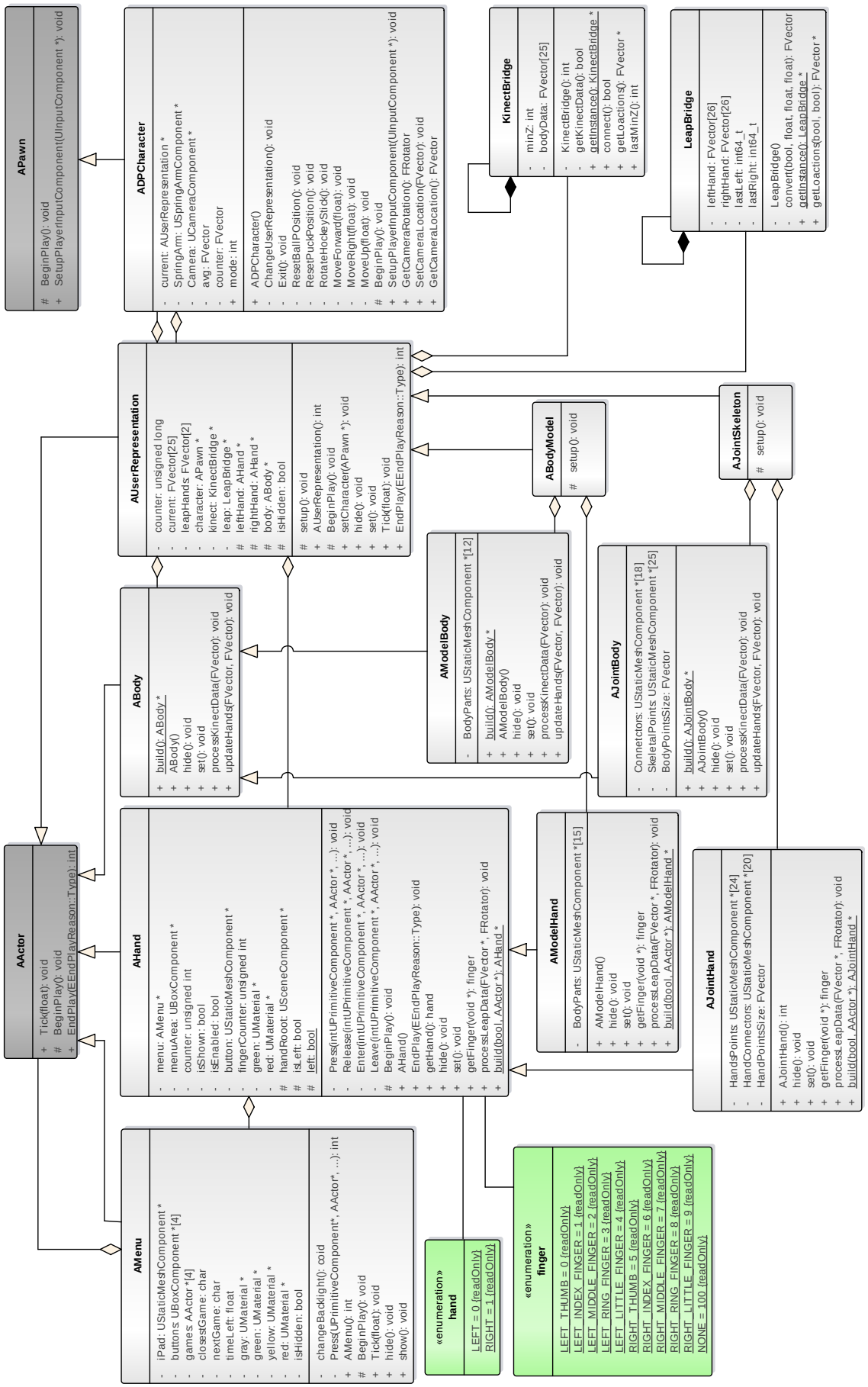

Figure 1: User Representation - UML Class Diagram

Also prefixes A and U in class names are generated by UE4. Classes inherited from *UObject* have prefix U and classes inherited from *AActor* have prefix A [1].

C.1. User Representation

User representation described in chapter 8.2. User Representation consists of 13 classes and two enumeration types (see Figure 1).

ADPCharacter

First class *ADPCharacter* (defined in DP/Source/DPCharacter.h and implemented in DP/Source/DPCharacter.cpp files) that extends EU4 class *APawn*. *APawn* class is meant as base class for representation of a player or AI.

ADPCharacter class contains user's camera which is automatically controlled by HMD and implements callbacks for key actions (see Appendix B.5. Keyboard Events).

To eliminate more noise effect from Kinect on camera position we apply average on each component of position if difference between current (average) and new position is less than 15.

KinectBridge & LeapBridge

Both classes follow *Singleton* pattern. Main purpose of these classes is to encapsulate logic needed for connection to device, getting new data from device and converting them to UE4 coordinates.

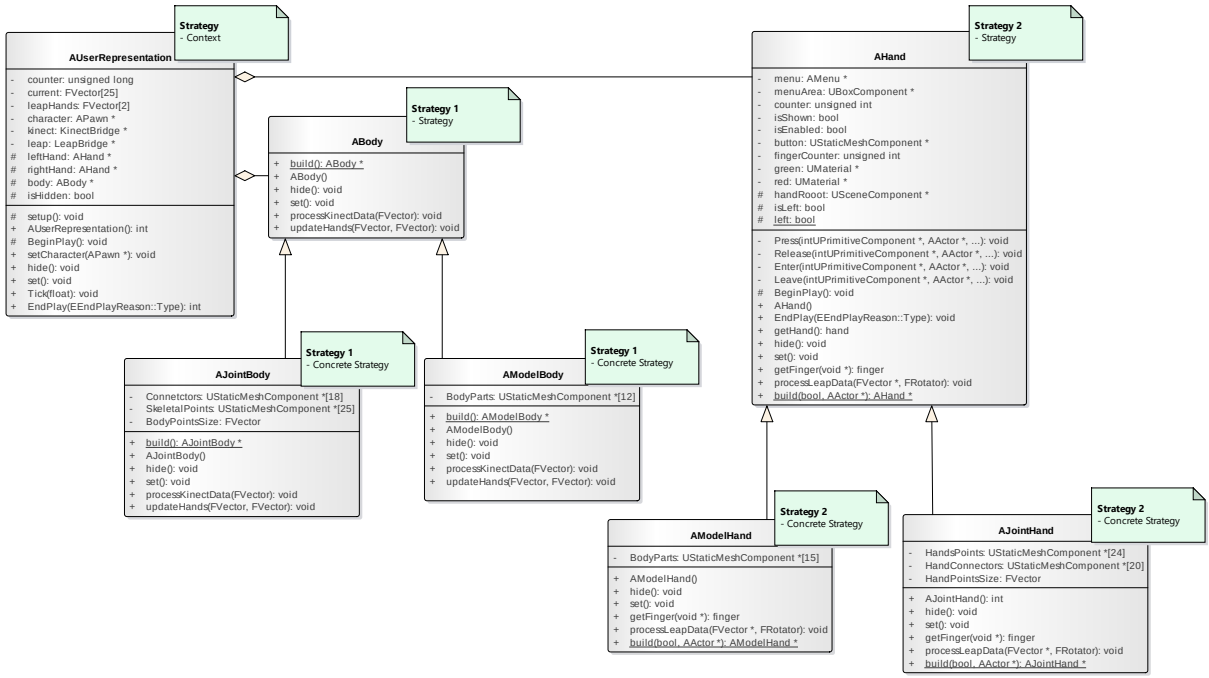

Figure 2: Strategy Pattern

[1] see https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard

AHand & ABody

Both classes represent strategy in strategy pattern. *AHand* contains one template function *processLeapData* which updates hand model's parts. *ABody* contains similar function called *processKinectData*. This function sets 3D body model parts except part between hand and elbow (see Source Code 1). Second template method is update hands where last two parts of model (parts between hand and elbow for both arms) are updated.

AHand additionally contains a "button" for opening and closing (functions *Press* and *Release*), logic for detection if there's enough space for menu to be open (functions *Enter* and *Leave*) and menu itself. The principle for detecting and pressing button is the same. Each uses its own counter that increases on entering collision shape and decreases on leaving. The main difference between pressing button and checking whether menu can be displayed is condition when counter changes. For pressing button, counter changers only when part of right hand enters or leaves and for detecting space it's when anything else than body or hand parts enters or leaves collision shape. In pressing button, action of displaying or hiding menu occurs when counter reaches value of zero, for detecting space when counter is above zero, button color changes to red, when decreases to zero - button color is changed back to green.

ModelBody.cpp

```
012  const char BodyEnds[BODY_PARTS_COUNT - 4][2] = {
013         { 12, 13 }, { 16, 17 }, { 13, 14 }, { 17, 18 },
014         { 14, 15 }, { 18, 19 }, {  4,  5 }, {  8,  9 }
015  };
016
017  const float BodyPartsHeights[BODY_PARTS_COUNT]{
018         55.0f, 55.0f, 45.0f, 45.0f, 20.0f, 20.0f,
019         22.0f, 22.0f, 31.5f, 49.0f, 31.5f, 31.5f
020  };
...
092     for (int i = 0; i < BODY_PARTS_COUNT; i++)
093         BodyParts[i]->SetRelativeLocation(data[BodyPartsPositionsMapping[i]]);
094
095     for (int i = 0; i < BODY_PARTS_COUNT - 4; i++)
096     {
097         BodyParts[i]->SetRelativeRotation(FRotationMatrix::MakeFromZ(
                                data[BodyEnds[i][0]] - data[BodyEnds[i][1]]).Rotator());
098         float ratio = FVector::Dist(data[BodyEnds[i][0]],
                                data[BodyEnds[i][1]]) / BodyPartsHeights[i];
099         BodyParts[i]->SetRelativeScale3D(FVector(0.85f, 0.85f, ratio));
100     }
```

Source Code 1: Update of body parts in function *processKinectData* of *AModelBody* class

Note that BodyEnds is 2D array where elements are indexes of Kinect Data array and elements [i][0] and [i][1] represents two endpoints of body part (e.g. value of [3][0] is 17 and [3][0] is 18. 17 and 18 refers to 17th and 18th element in Kinect data – Right Knee and Right Ankle). *BodyPartsHeights* is an array of model parts heights at scale 1.

AUserRepresentation

Class that contains main logic for updating user representation as described in chapter 8. Solution. Only function that needs to be overridden is protected virtual function *setup()*. The

purpose of this function is to setup – create instances for Hands and Body (see Source Code 2). Other function s like *hide()*, *set()* or *EndPlay()* can also be overridden in subclasses to modify behavior of given functions if needed.

JointSkeleton.cpp

```
007  void AJointSkeleton::setup()
008  {
009          body = AJointBody::build(this);
010          leftHand = AJointHand::build(true, this);
011          rightHand = AJointHand::build(false, this);
012  }
```

BodyModel.cpp

```
007  void ABodyModel::setup()
008  {
009          body = AModelBody::build(this);
010          leftHand = AModelHand::build(true, this);
011          rightHand = AModelHand::build(false, this);
012  }
```

Source Code 2: Implementation of *setup()* function in *AJointSkeleton* & *ABodyModel* classes

Update of user representation is performed in function *Tick()*. This function is called every frame.

After checking whether devices are available and user representation is shown, we increase counter of frames, where LM data hasn't been received (see Source Code 3 – line 70).

AUSerRepresentation can be used without A*DPCharacter* class. In this case no instance of *ADPCharacter* has been assigned to it. When instance has been assigned by calling function *setCharacter()*, we get actor's camera current location and rotation (see Source Code 3 – lines 73 - 80).

```
066  void AUserRepresentation::Tick(float DeltaTime)
067  {
068     // Check if User representation is displayed and devices are available
070     counter++;
071
072     // Get Camera Location & rotation
073     FRotator cameraRot = FRotator(0.0f, 0.0f, 0.0f);
074     FVector cameraLoc = FVector(0.0f);
075
076     if (character)
077     {
078         cameraRot = ((ADPCharacter *)character)->GetCameraRotation();
079         cameraLoc = ((ADPCharacter *)character)->GetCameraLocation();
080     }
```

Source Code 3: *AUserRepresentation Tick* - first steps

Next step is to get LM data for each hand and check if data represents required hand (see Source Code 4). Since 3D models can vary depending on left and right hand, and for us to be able to properly connect hands with elbow, we need to make sure that data represents proper hand.

Based on data for which hand is available, validation can be performed in three cases: for both hands, left hand only or right hand only (see Source Code 4 lines 87, 105 and 118).

First step of validation is summation of camera position and rotated hand position from LM data (see Source Code 4 lines 89-90). This calculation gives us locations in the same relative coordinates as Kinect data are. Than we calculate all four passible combinations of distances (Kinect data at index 7 are for left hand and index 11 for right hand). If LM left hand is closer to Kinect Right hand and LM right hand is closer to Kinect Left hand, we swap LM data (see Source Code 4 lines 97-102). Since we only need to know which pairs of points are closer, not their exact distances, we use squared distances to speed up calculation.

Lastly since we have data from LM, we set number of frames without LM data to zero.

The same process is performed when we have data only for one hand. In these cases, we perform only steps for one hand (e.g. lines 89, 92 and 93 for left hand). Swapping with one hand ocours only if LM left hand is closer to Kinect Right hand or LM right hand is closer to Kinect left hand.

```
083     FVector *leftHandData = leap->getLoactions(true, true);
084     FVector *rightHandData = leap->getLoactions(false, true);
085
086     // Check if data given by Leap represents given hands
087     if (leftHandData && rightHandData)
088     {
089         FVector leapL = cameraLoc + cameraRot.RotateVector(leftHandData[0]);
090         FVector leapR = cameraLoc + cameraRot.RotateVector(rightHandData[0]);
091
092         float LLdist = FVector::DistSquared(leapL, current[7]);
093         float LRdist = FVector::DistSquared(leapL, current[11]);
094         float RLdist = FVector::DistSquared(leapR, current[7]);
095         float RRdist = FVector::DistSquared(leapR, current[11]);
096
097         if ((LRdist < LLdist) && (RLdist < RRdist))
098         {
099                 FVector *tmp = leftHandData;
100                 leftHandData = rightHandData;
101                 rightHandData = tmp;
102         }
103         counter = 0;
104     }
105     else if (leftHandData) { ... }
118     else if (rightHandData) { ... }
```

Source Code 4: *AUserRepresentation Tick* - Getting LM data and validating them

After validation of data, we update state of each hand for which data is available (see Source Code 5). We also update hand position (position relative to LM sensor).

```
132     // Process Left Hand
133     if (leftHand && leftHandData)
134     {
135         leftHand->processLeapData(leftHandData, cameraRot);
136         leapHands[0] = leftHandData[0] + FVector(5, 0, 0);
137     }
138
139     // Process Right Hand
140     if (rightHand && rightHandData) { ... }
```

Source Code 5: *AUserRepresentation Tick* - updating of hands state

After processing LM data, next step is processing of Kinect data.

If Kinect provides new data, we calculate offset of which we need to move up Kinect data. Values of Kinect data depend on height on which sensor is physically placed. To take into account different heights we calculate offset of which we move up or down Kinect data values, so user can stay on the ground and not fly or collide with it (see Source Code 6 lines 152 and 157).

To eliminate noise effect, we calculate distance between old and new position and if it is less than threshold ($\sqrt{7}$ cm) old position will be used (see Source Code 6 lines 159 - 160).

After checking and modifying data, we update body representation and camera location (see Source Code 6 lines 162 and 165).

```
148  FVector *kinectData = kinect->getLoactions();
149
150     if (kinectData && body)
151     {
152         float Zoffsef = -(kinectData[kinect->lastMinZ()].Z + 98);
153
154         // Move everything up
155         for (int i = 0; i < KINECT_JOINTS_COUNT; i++)
156         {
157             kinectData[i].Z += Zoffsef;
158
159             if (FVector::DistSquared(kinectData[i], current[i]) >= KINECT_TRESHOLD)
160                       current[i] = kinectData[i];
161         }
162         body->processKinectData(current);
163
164         if (character)
165             ((ADPCharacter *)character)->SetCameraLocation(current[3] + ... );
166     }
```

Source Code 6: *AUserRepresentation Tick* - Processing of Kinect Data

Lastly, we set hands positions and connect them with body. Depending on which condition is met, setting hand position is based on LM or Kinect data (see Source Code 7 lines 169 - 172).

```
169     if (leftHand && leftHandData)
170       leftHand->SetActorLocation(GetActorLocation() + GetActorRotation().
                RotateVector(cameraLoc + cameraRot.RotateVector(leapHands[0])), false);
171     else if ((leftHand && rightHandData) || (kinectData && (counter > 10)))
172       leftHand->SetActorLocation(GetActorLocation() + GetActorRotation().
                RotateVector(current[7]), false);
173
174     if (rightHand && rightHandData)  ...
178
180     if (body && leftHand && rightHand)
181       body->updateHands(leftHand->GetActorLocation(),rightHand->GetActorLocation());
182  }
```

Source Code 7: *AUserRepresentation Tick* – Updating hands positions and connecting hands with body

C.2. Taking Objects

Class *UTakeHandler* encapsulates logic for taking objects with one or both hands. What will be applied on concrete object depends on settings defined By *TakeRules*.

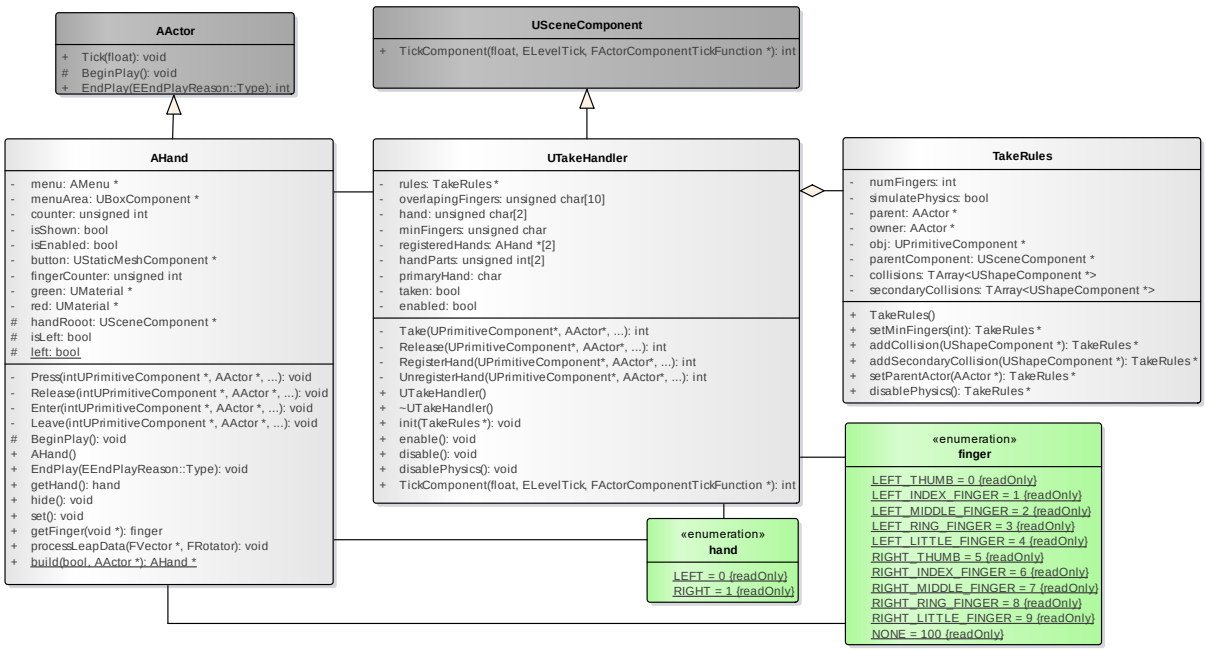

Figure 3: UML – Class diagram for interaction

TowerOfHanoiDisk.cpp

```
060  rules = new TakeRules(disk, collision[0]);
061
062  for (int i = 1; i < 4; i++)
063          rules->addCollision(collision[i]);
```

HockeyStick.cpp

```
037  TakeRules *rules = (new TakeRules(stick, top))->addSecondaryCollision(bottom)->
                                                                    disablePhysics();
```

Source Code 8: Creation of rules in *ATowerOfHanoiDisk* and *AHockeyStick* constructors

TakeRules

Class *TakeRules* (see Figure 3) encapsulates settings for performing take action.

setMinFingers(int) - If for taking given object is required to hold it with more than two fingers, argument specifies how many fingers needs to be detected before taking object. Accepted value is in range between two and five, including both of them.

*addCollision(UShapeComponent *)* - sets additional collision detection shape. This area represents place, where user can take object (see Source Code 8 - TOH disk contains 4 collision boxes for taking it).

*addSecondaryCollision(UShapeComponent *)* - sets collision detection shape for second hand. This collision area has to have all primary areas (set in *Constructor* and by calling *addCollision()*).

*setParentActor(AActor *)* - sets parent actor object to which will be reattached after releasing.

disablePhysics() - disables automatic simulating physics on releasing of object.

UTakeHandler

Handles Take actions specified by *TakeRules*.

After creating instance of UTakeHandler, it is necessary to call *init* function (see Source Code 9). Init function sets callbacks for collision shapes.

```
058  takeHandler = CreateDefaultSubobject<UTakeHandler>(TEXT("UTakeHandler"));
...  ...
065  takeHandler->init(rules);
```

Source Code 9: Creation and initialization of *UTakeHandler* in *ATowerOfHanoiDisk* Constructor

Class contains four callbacks functions:

- *Take* – entering primary area (one hand)
- *Release* – leaving primary area (one hand)
- *RegisterHand* – entering secondary area (two hands)
- *UnregisterHand* – leaving secondary area (two hands)

All four functions are called always when object enters (*Take* and *RegisterHand*) or leaves (*Release* and *UnregisterHand*) collision area. First step in all four cases is to check if object that entered area is part of a hand.

In *Take* and *Release* we check which finger entered or leaved area. If it is other part of hand than finger, we exit function (see Source Code 10).

```
012  051     if (rules == nullptr)
013  052         return;
014  053
015  054     if ((OtherActor != nullptr) && (OtherComp != nullptr))
016  055         if (OtherActor->GetClass()->IsChildOf(AHand::StaticClass()))
017  056         {
018  057            finger f = ((AHand *)OtherActor)->getFinger(OtherComp);
019  058            if (f == NONE)
020  059                return;
```

Source Code 10: Initial check in *Take* and *Release* functions

After getting finger, we increase or decrease count of objects of given finger (see Source Code 11 lines 22 and 61). If first part of finger enters or last part of finger leaves the area, we check to which hand finger belongs and increase or decrease counter of fingers for given hand (see Source Code 11 lines 24-27 and 63-66).

Next check is for condition of taking or releasing object. Condition consists of checking minimal number of fingers in area of one hand and whether thumb of same hand is present in area as well.

```
022     overlapingFingers[f]++;
023
024     if (overlapingFingers[f] == 1)
025     {
026         char h = ((AHand *)OtherActor)
                              ->getHand();
027         hand[h]++;
028
029         if (!enabled)
030            return;
031
032         if (hand[h] >= minFingers &&
                overlapingFingers[5*h]>0
                && !taken)
033         { /* see Source Code 12 */ }
```

```
061     overlapingFingers[f]--;
062
063     if (overlapingFingers[f] == 0)
064     {
065         char h = ((AHand *)OtherActor)
                              ->getHand();
066         hand[h]--;
067
068         if (!enabled)
069            return;
070
071         if (hand[h] < minFingers ||
                overlapingFingers[5*h]==0
                && taken)
072         { /* see Source Code 13 */ }
```

Source Code 11: *UTakeHandler Take* (left) and *Release* (right) – hand & finger processing

When condition for taking object is met, we disable physics. For object to be able to move with the hand we also need to reattach the object to parent component (in UE4 after enabling simulating physics on component, component automatically detaches from parent). Lastly, we attach the whole hierarchy of components to the hand.

If object can be held by two hands, we mark current hand as primary.

```
034         taken = true;
035         rules->getObject()->SetSimulatePhysics(false);
036         rules->getObject()->AttachToComponent(rules->getParentComponent(), ... );
037         rules->getOwner()->AttachToActor(OtherActor, ... );
038
039         if (registeredHands[0] != nullptr)
040                 if (OtherActor == registeredHands[0])
041                         primaryHand = 0;
042                 else if (OtherActor == registeredHands[1])
043                         primaryHand = 1;
```

Source Code 12: *UTakeHandler Take* - Taking object

On releasing object, firstly we detach object from the hand. If object has defined parent actor, we reattach the object hierarchy to parent Actor. Lastly, we set simulating physics based on value defined in rules.

```
073         taken = false;
074         primaryHand = -1;
075         rules->getOwner()->DetachRootComponentFromParent(true);
076
077         if (rules->getParentActor() != nullptr)
078                 rules->getOwner()->AttachToActor(rules->getParentActor(), ... );
079
080         rules->getObject()->SetSimulatePhysics(rules->isSimulatingPhysics());
```

Source Code 13: *UTakeHandler Release* - Releasing Object

For second hand we remember how many objects of given hand entered area and reference to hand that is inside. When first element of hand enters area (see Source Code 14 lines 93-94), we save reference to hand and when last element of hand leaves, we set reference to *nullptr* (see Source Code 15 lines 112 - 114).

```
092         char h = ((AHand *)OtherActor)->getHand();
093         if (registeredHands[h] == nullptr)
094                 registeredHands[h] = (AHand *)OtherActor;
095         if (registeredHands[h] != OtherActor)
096         {
097                 registeredHands[h] = (AHand *)OtherActor;
098                 handParts[h] = 0;
099         }
100         handParts[h]++;
```

Source Code 14: *UtakeHandler RegisterHand*

```
108         char h = ((AHand *)OtherActor)->getHand();
109         if (registeredHands[h] != OtherActor)
110                 return;
111
112         handParts[h]--;
113         if (handParts[h] == 0)
114                 registeredHands[h] = nullptr;
```

Source Code 15: *UTakeHandler UnregisterHand*

C.3. Use- Cases

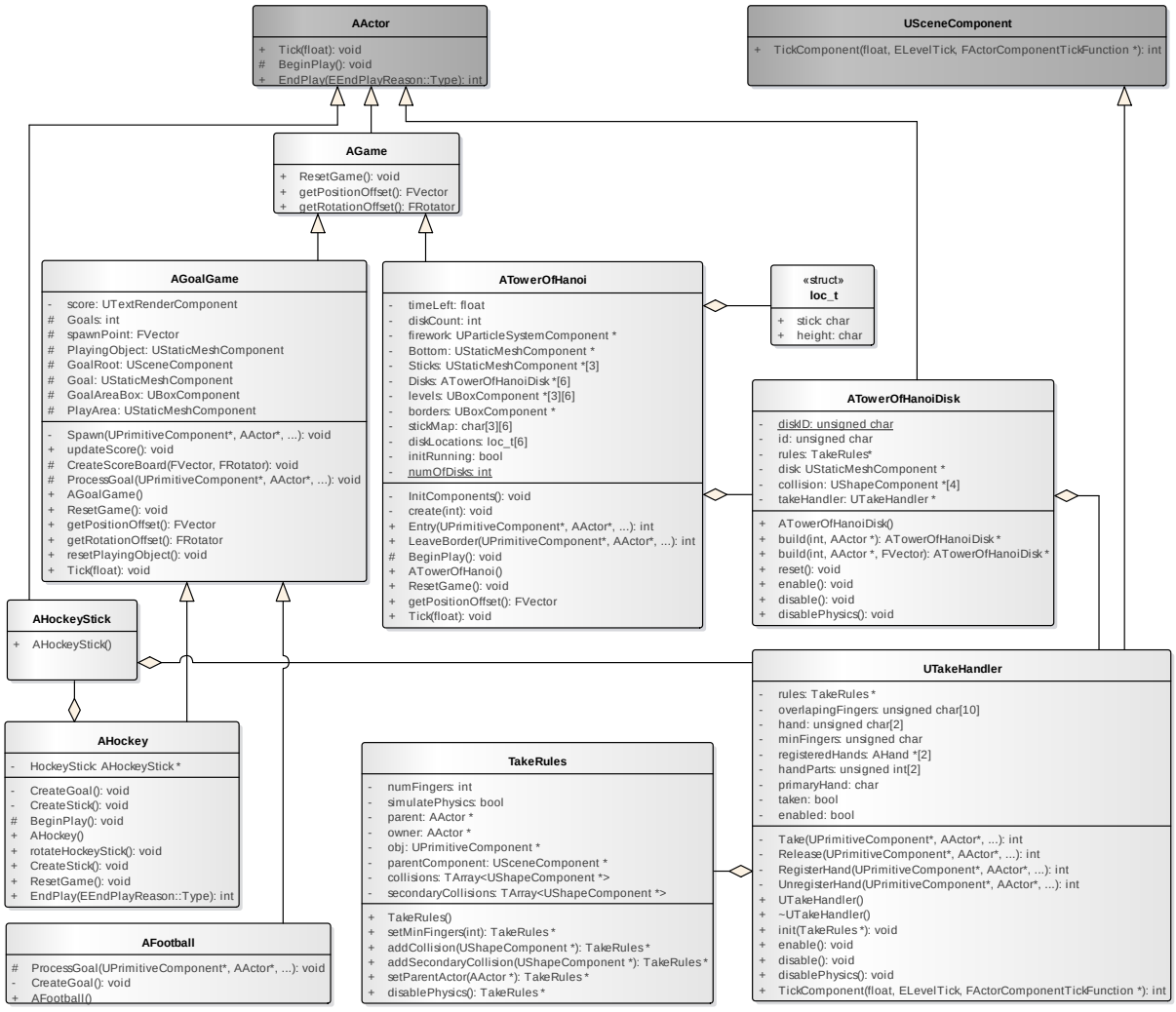

Figure 4: UML Class Diagram for implemented Use-Cases