8. Solution

We implemented our solution using UE4 and MS Visual Studio 2017. In implementation we used C++. In order to access data from Kinect v2 and LM sensor, we used Kinect for windows C++ SDK and Leap SDK (not UE4 plugin).

Since the main focus of UC1 (Playing game of Tower of Hanoi) and UC2 (solving Jigsaw puzzle) is based on the same aspects (moving objects), we decided to continue with implementation only with UC1.

8.1. Setup

Used setup contains four main parts:

- Computer (Lenovo laptop),
    - Intel Core i7-6700HQ with 16GB of memory
    - NVidia GeForce GTX 1070 8GB
- Oculus Rift DK2 HMD,
- MS Kinect v2 Sensor and
- LM Sensor mounted on HMD.

For this setup and project to work properly is necessary to make sure that following conditions are met:

1. LM controller is mounted on HMD in way, that light indicator is facing downwards.
2. The distance between Kinect and user is more than minimal distance and less than 4.5 meters.
3. Before connecting HMD to computer, it's important that HMD is facing Kinect sensor.

Since we know that vertical FOW for Kinect is 60° [25] we can create formula to calculate minimal distance from the sensor:

In most cases, Kinect sensor is not placed at height that is equal to half of user's height. Therefore, we need to calculate distances for highest and lowest points of user`s body. Minimal distance from Kinect is then maximal value from both distances. It can be expressed mathematically in following formula:

$$distance(h) = \frac{h * \cos 30°}{\sin 30°} = h * \cot 30°$$

$$minimal\_distance(h_k, h_u) = \max\left(distance(h_u - h_k), distance(h_k) \right)$$

Where h_k is height at which is Kinect placed and h_u is height of user.

To be more precise with calculation of minimal distance, we should also include in our calculations horizontal FOW (which for Kinect v2 is approximately 70°) and maximal width of user. Distances could be calculated in a similar way.

8.2. User Representation

For user representation we used data from both Kinect and LM sensors. We implemented (for more information about implementation see **Error! Reference source not found.**) two models of user representation:

1. Skeleton points – created using spheres and cylinders
2. Body model – using models of body parts.

Firstly, we tried to use rotation (quaternions) information from Kinect. Getting rotation information from Kinect data and applying it to built-in 3D model. This approach required creation of new C++ class and new UE4 blueprint. In C++ we were getting data from Kinect sensor and in blueprint setting values. We came to difficulties with converting quaternions from Kinect space to UE4 space. Since we spent on this approach more time then we initially planned, we decided to abandon this approach and use position information instead.

Since all three of used technologies (UE4, Kinect and LM) use differently oriented axis, first step after receiving new data from sensors was to convert them to UE4 space.

In Kinect case we're doing simple mapping:

$$Location_{UE4}(x, y, z) = Location_{Kinect}(-z, x, y) * 100$$

Multiplication by 100 is needed because Kinect location data are in meters but UE4 uses centimeters.

For LM sensor mounted on HMD with light indicator facing down and facing Kinect (zero rotation angle of HMD):

$$Location_{UE4}(x, y, z) = \frac{Location_{LM}(y, x, z)}{10}$$

Since LM is physically mounted on HMD and its orientation (rotation of axis) depends on what user is currently looking at, additional application of camera rotation was necessary.

Update of user representation model is preformed every frame (see Figure 1 - Tick). First sensor that we check for new data is LM (see Figure 1 – 2. Get Hand Data from LM). If new data are available at least for one hand, we validate them with latest Kinect data (see Figure 1 – 3. Check distances between LM and Kinect Hand Data).

LM often classifies left and right hand wrongly, therefore we need to validate them to Kinect data. First step of validation is to apply camera rotation to LM hand location data. Then rotated new coordinates sums with camera location. After this we should have approximately LM hand location data and Kinect data in same relative coordinates. Then we compare distances between new LM hand positions and Kinect Hand positions. If LM left hand is closer to Kinect right hand and LM right hand is closer to Kinect Left hand, we swap LM hand data (see Figure 1 – 4. Swap Hand Data). If ML has data only for one hand, we assignee them to hand, which are closer relative to Kinect data. After LM data validation, we update state of

each hand, for which we have available data (see Figure 1 – 5. Update Left Hand State & 6. Update Right Hand State).

After updating hands, we check for new data from Kinect (see Figure 1 – 7. Get Body Data from Kinect).

Kinect provides new data at speed of 30fps. HMD refresh rate needs to be at least 90fps. This means that approximately at each third frame we get new data from Kinect. This is also supported by our observation (e.g. by logging whether new data are available or not). We observed real delay between new data at rate between two and five frames. During development we also observed inaccuracy and noisiness of data provided by the Kinect. To partially eliminate negative effect of noise (random movement around real position), we applied threshold for acceptance of a new value. If distance between new and old position of point is more than threshold, new position will be applied, otherwise we will ignore new position. This approach partially eliminated random back and forth movement produced by the noise. Application of some more advanced noise-reduction algorithm could be more appropriate. Also applying some algorithm for user's position prediction could be applied when Kinect doesn't have new data available. Both algorithms (noise reduction and prediction) mustn't be slow that will produce drops in framerate.

In our case, we update body state only if new data is available (see Figure 1 – 8. Update Body State). This update does not include part between elbow and hand.

Our firs approach to connect hands with the rest of body was to set hand (root) position depending on data hand position information from Kinect data. This approach caused delays in hand movement. Therefore, for our final solution we chose different approach.

Depending on availability of new LM data we have three cases:

1. Data for given hand are available,
2. Data for given hand aren't available but for other hand are,
3. No new data are available.

If for given hand LM data has been provided, we update position of hand (see Figure 1 – 9. Update Left Hand Position Based on LM Data and 11. Update Right Hand Position Based on LM Data) by summation of LM hand root position and camera position. If for given hand aren't available LM data but for other hand are (e.g. LM detects only Left hand), we update position of hand, for which we don't have LM data by information provided by Kinect as described in first approach hand (see Figure 1 – 10. Update Left Hand Position Based on Kinect Data and 12. Update Right Hand Position Based on Kinect Data). In last case, when no LM data have been provided for both hands, we increase counter of missing frames. When this counter accedes value of 10, we update hand position based on Kinect data. When we don't have any new LM data, we do not automatically update based on Kinect data, because Kinect data are inaccurate and since LM processing and framerate aren't in sync, there's possibility that current frame has been produced faster than LM processing of new data and making them available for applications. We do not want to introduce additional noisiness by always switching between LM and Kinect mapping. On the other side, when we got only data for one hand, we know that other hand is not in view point of LM, therefore user is not looking at

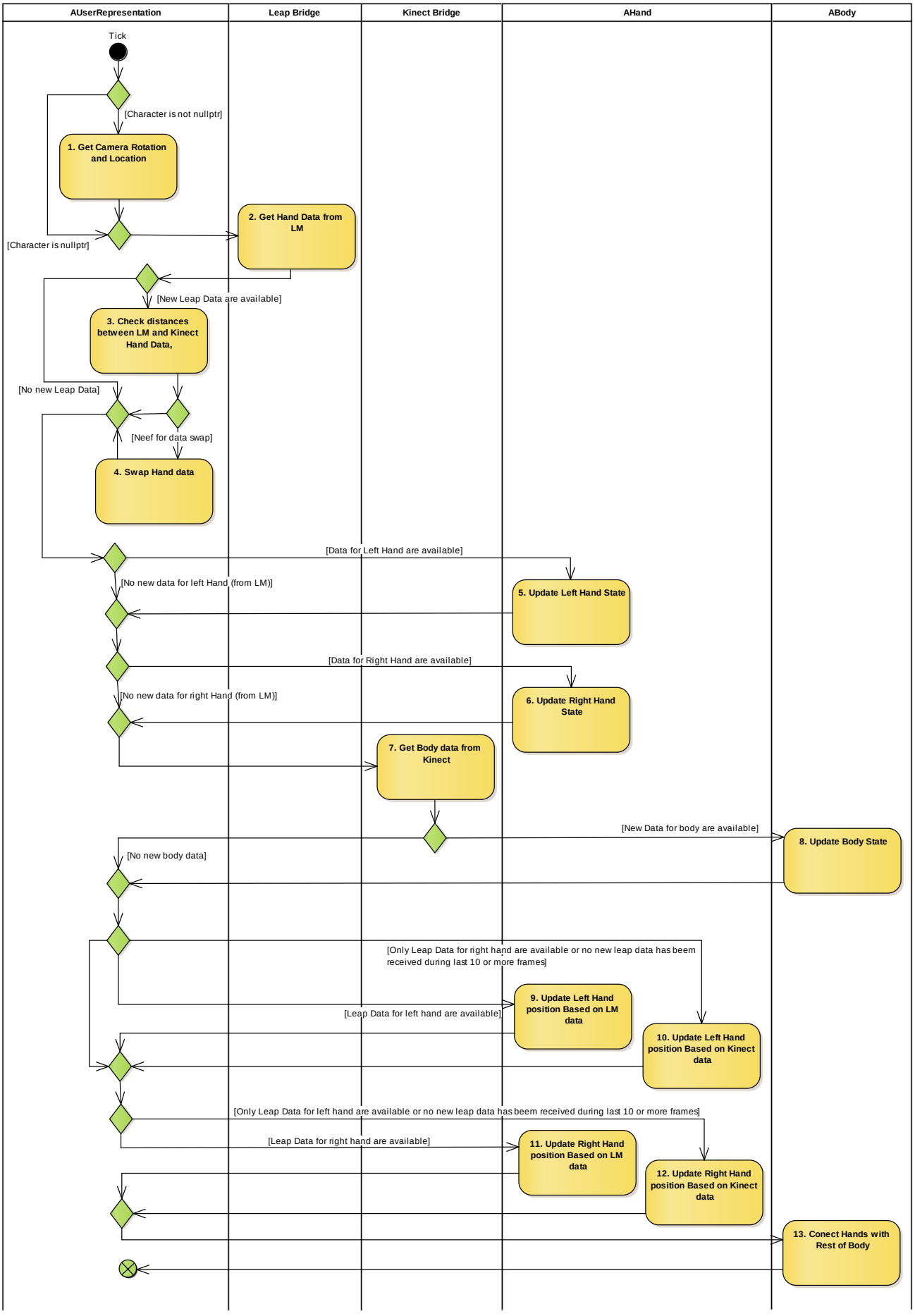

Figure 1: Update of user Representation (UML - Activity Diagram)

given hand. Leaving hand on old position when user moved is unnatural.

Last step of updating user's representation is to connect elbow with hand based on their current positions (see Figure 1 – 13. Connect Hands with Rest of Body). This update is performed only when some new data has been processed.

8.3. Interaction

UE4 offers only simple shapes to be used as a collision area. These shapes include: box, sphere and capsule.

Since TOH disk has more complex shape, we added multiple (four) box collisions around it to better match its shape (see Figure 2 left). For each of collision boxes we registered the same callback functions on the same object. Firs callback is when other object enters given area and second when eaves this area.

Class that contains take logic (class *UTakeHandler*, for more details of implementation see Technical Documentation in Appendix C.2. Taking Objects) has 10 elements array of numbers with all elements initialized to zeroes. Each element represents one finger. Value inside of this array represents how many objects for given finger are inside of area (area bounded at least one collision box). In callback for entering this area we check which finger part entered it and increase given element of array. Similar action takes place in callback for leaving, where we decrease value for given finger. If object that doesn't belong to any finger enters or leaves this area, object is ignored. Our attention is when value in array increases over zero or decreases to zero. When this happens, we check condition for taking object (thumb and minimal fingers of given hand). Based on result of condition check, we attach or detach TOH disk from hand.

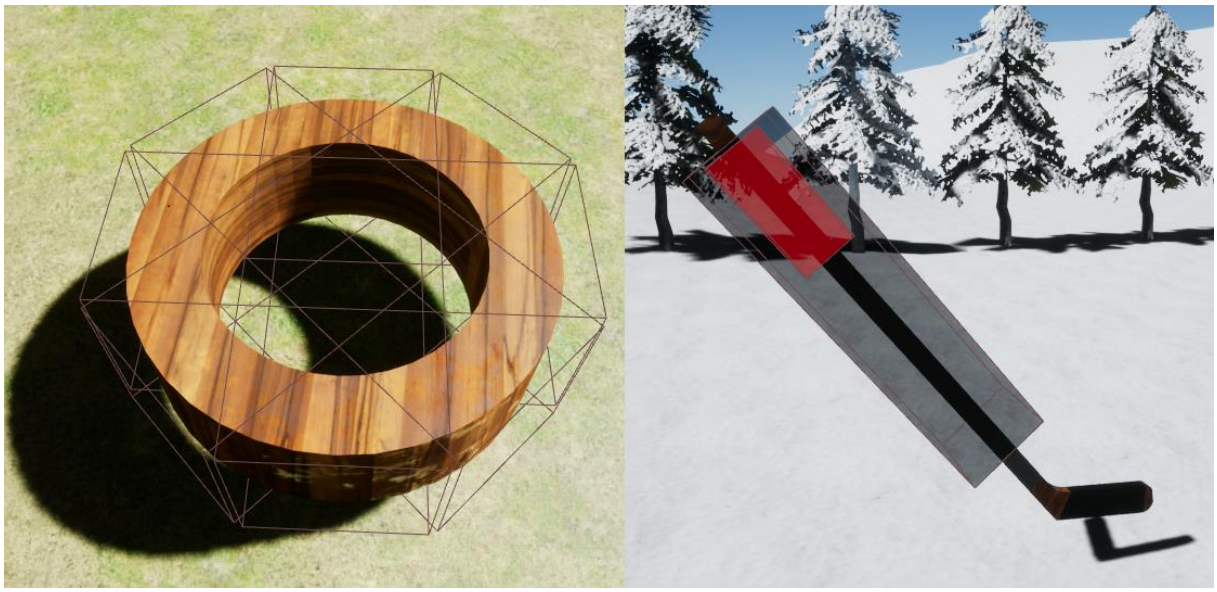

Figure 2: TOH disk (LEFT) and Hockey Stick (right): primary collision box (red) and secondary collision box (black)

For hockey stick we re-used taking from TOH disk (see Figure 2 right – transparent red

box). For hockey stick we added secondary collision area (unlike with TOH Disk, both collision boxes in hockey stick has different functionality). Secondary area (see Figure 2 right – transparent black box) fully includes primary area. This area detects only whether hand is inside or not (not fingers). For detecting hands, we use similar principle (increasing counter on entering area and decreasing on leaving area) like for fingers but only array with two elements. If secondary area detects both hands and hockey stick is attached to hand by actions of primary area, we calculate new position and rotation for hockey stick.

First step of updating position and rotation of hockey stick is calculating distance between hockey stick and hand, that stick is attached to (i.e. hand detected by primary area). Secondly, we calculate ray-sphere intersection. Ray that starts in primary hand (hand, which hockey stick is attached to) and its direction is to second hand. The origin of sphere is at primary hand (ray and sphere have the same origin) and radius is of distance calculated between hand and hockey stick. The point of ray-sphere intersection is new position of hockey stick. Finally, we set rotation of hockey stick that hockey stick heel is closer to second hand (i.e. hand not inside of primary area).

8.4. Use-cases

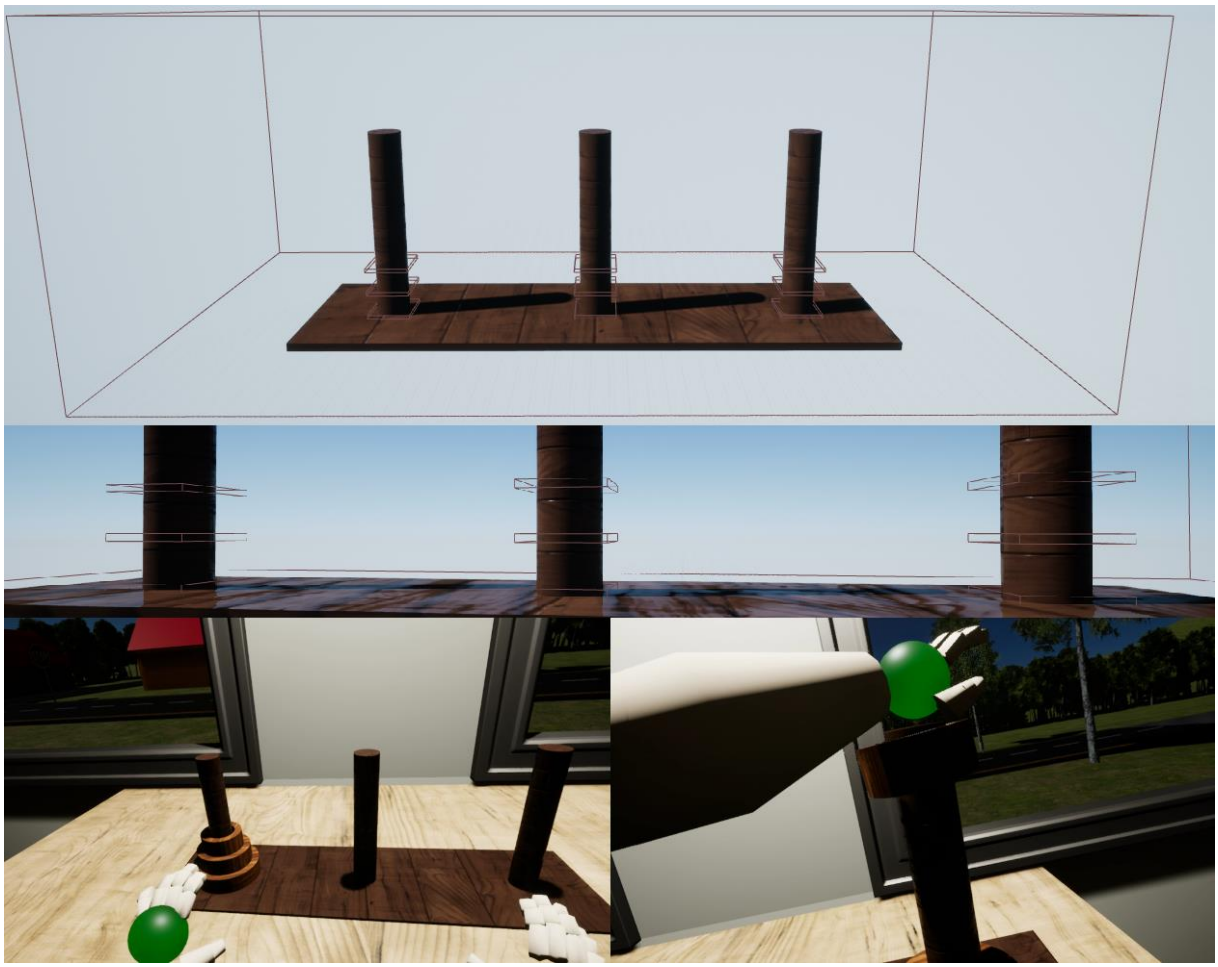

Figure 3: TOH: Game before start (top and middle) and solving game in VR (bottom)

Around all three games we put collision boxes (see Figure 3 - top and Figure 4) to detect when object (ball, puck or disk) is too far for user to interact with. When object leaves this are, is automatically spawned on pre-defined position. For hockey and Football this position is always the same (start position), for TOH it is the last location where disk was placed (from where was taken).

At the beginning of TOH, all disks are placed on far-left stick. All disks except the one at the top are blocked – user can't move or take them. When top disk is placed on different stick, disk that has been under it is enabled and disk that is now under it is blocked. Only the top disk at each stick is enabled.

We also added small collision boxes to detect location of a disk (see Figure 3 – top and middle) and for each disk we remember its position (which disk at which height). In these areas we perform multiple check to determine validity of move:

- Is some disk already registered there?
- What disk is under given location (if current location is not zero)?
- Which disk are we trying to place there?

If move is valid – placing smaller disk on larger or empty stick, we update location of disk. In case that location under is empty (placing disk on empty stick and first area that detects it is the most top one) we ignore call. If user tries to place larger disk on smaller one, we spawn smaller disk on its original location.

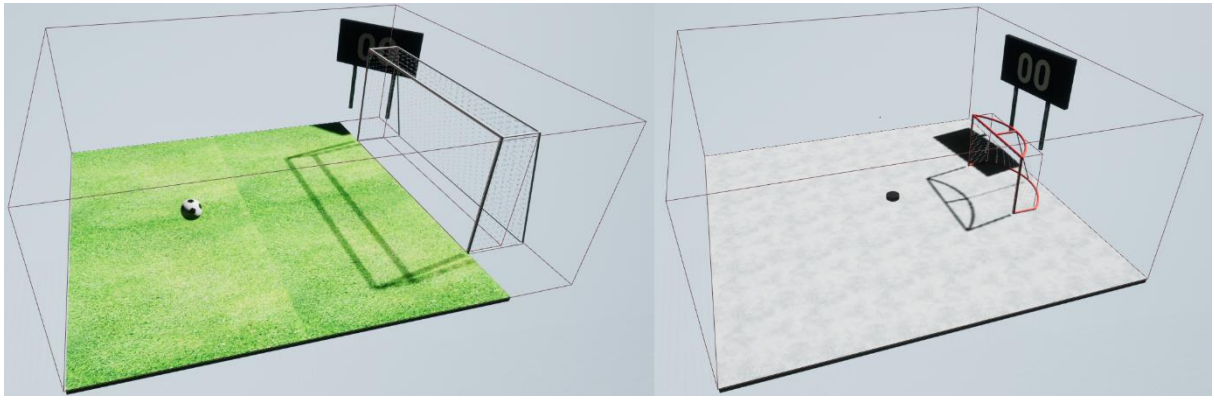

Figure 4: Football Field (left) and Hockey area right)

Parts that both football and hockey share are processing goal and leaving of area, play area and score board. Goal is detected when playing object (ball or puck) enters collision box inside of 3D goal model.

Parts that are set differently for each game include:

- 3D models of goal and playing object (i.e. ball or puck),
- Scale of enclosing area and goal.
- Location of goal and playing object initial (and spawn point)
- Material for play area.

Hockey additionally contains a hockey stick.

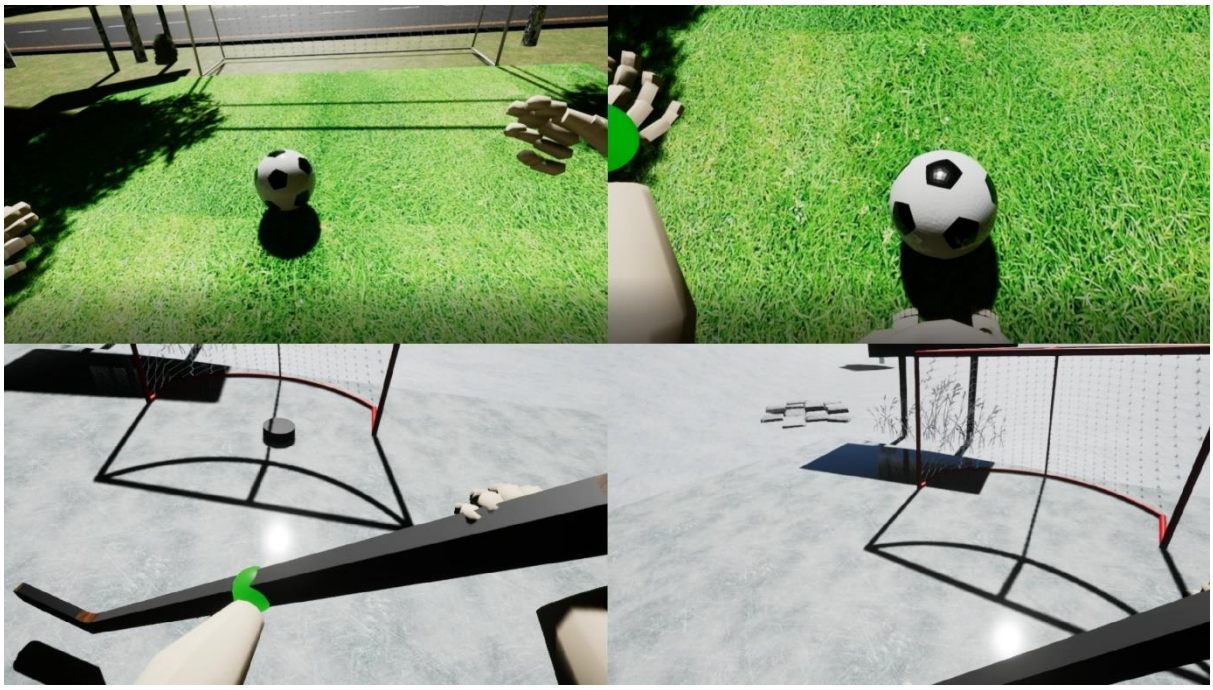

Figure 5: View from VR: football (top) and hockey (bottom)

8.5. Virtual environment

Figure 6: VE: House and Garden (left top), view from garden to mountains (right top), VR room (left bottom) and Mountains hockey (right bottom)

We created VE as described in chapter 6.3. except hockey, which we placed I nearby mountains. Since house and garden are in green (summer) environment, place where we could expect ice and snow are mountains.

House is surrounded by forest, mountains and other houses.

8.6. Sources

In our solution we used third-party 3D models:

- (Summer) Vegetation pack[1] (Summer vegetation models):
    - https://drive.google.com/file/d/0B44B2m5zm7l2UlR4Vkl2YU9sWW8/view
- Materials:
    - https://www.poliigon.com/
- And "Infinity Blade: Ice Lands" pack[1] (free from UE4 marketplace) and other UE4 Content.
- The rest of models have been created with blender 2.78b.

[1] Unused parts of pack have been removed from final project