Visualization of a robotic arms working area

Flückiger Quentin (flucq1@bfh.ch)

January 18, 2019

Contents

1 Introduction

Work, this activity most of us must do daily to earn a salary and get on with our lives and hobbies. During this activity we will have to deal and work with co-worker, we start to understand how people behave at our workplace and even maybe in general for this sort of work. The more we get used to the way our co-workers work, and they get used to how we work, the more we can predict what will be there actions and so anticipate them to be more efficient. But with the appearance of robot, taking more and more subtasks, it started to be difficult to predict their action as they have a complete different perception as we do, if we can call it perception. And with every update their behaviour would vary so our time getting used to them would go to waste and we would start over.

Let's take for example a robot on a montage chain taking a piece "A" to put it on the chain, and a human employee in front of it. This robot does the same pathing with its arm for weeks, but on a Monday morning after an update it has a new pattern. Unfortunately, the employee is unaware of this. He lets his glasses slip close to the robot, lucky him they are on the side where the robot never goes, so he reaches out his hand to grab them. And at this time the robot used one of the new pattern he was just improved with in the new update. Wouldn't it be great to have a way to visualize and predict the movement of such a robot?

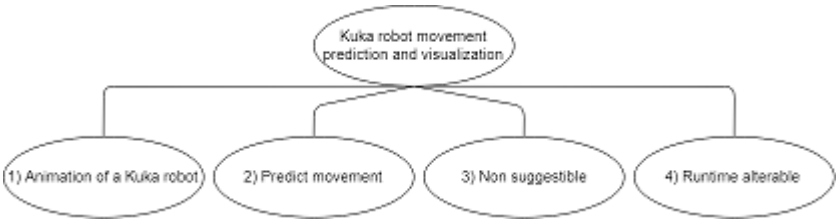
Well this project will cover the topics movements visualization of a robotic arm and prediction about its whereabout.

2 Project requirements

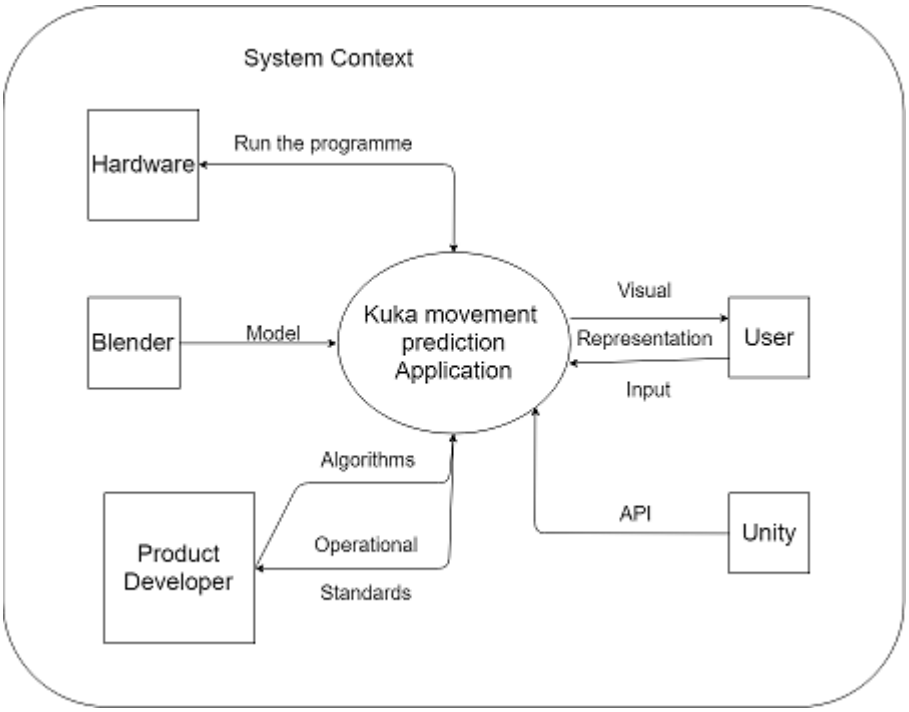
2.1 Vision

The project Visualization of a robotic arm's working area is aimed to create a software which will simulate the movements of a Kuka robot. It will then represent the robots' current displacement and its future possible location.

2.2 Goals



2.3 System context



3 Phase 1

3.1 Environment setup / Modelling

In the first step of phase one came all the setup of the different software, API, system and models to be used during the rest of the project. I decided to use Unity to develop the application for the following reasons. I am already a little bit familiar with it, it is cross platform, the documentation is abundant, and it is rather easy and intuitive to take in hand. As the use of GitLab was suggested by Mr. Fuhrer I went with this solution to share my work and progress. But for Git to recognize the change in Unity files we need a special .gitignore. This .gitignore can be added when we create a repository with GitHub, so I decided to create the repository with GitHub and mirror it on GitLab. For this to work, we need to do change two settings in Unity editor settings. A step by step guide on how to setup github with unity can be found here https://www.studica.com/blog/how-to-setup-github-with-unity-step-by-step-instructions . The use of LaTex was strongly recommended as well, as it allows to see exactly what has been change in the documentation file, contrary to a word file for example which will erase the old one and push the new one every time a commit is done. To be able to use LaTex I had to instal Visual Studio Code, LaTex Compiler and Texlive. I had to solve a little issue where I was trying to compile with pdfTex when I had LuaLaTex. Afterward I looked for a free, open source 3D model of a Kuka arm robot. I came up with three solutions.

1

2

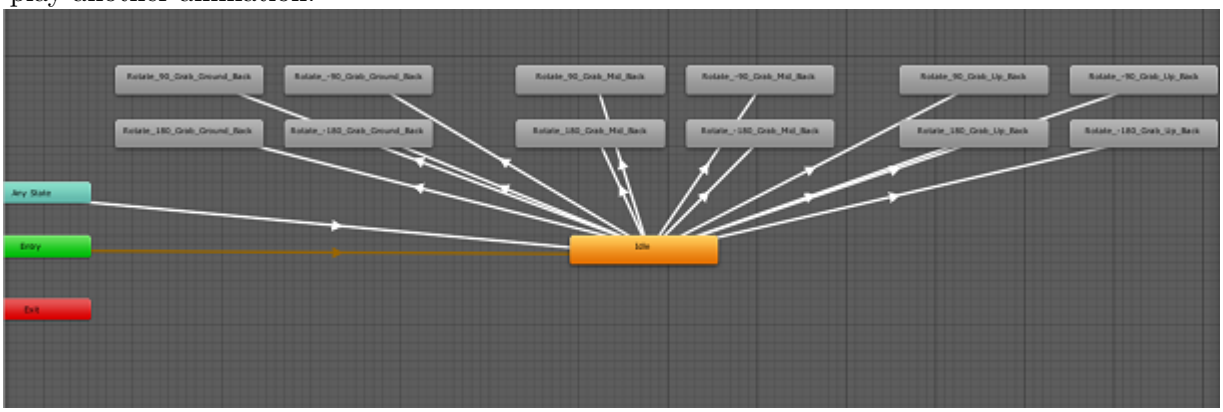
3

I chose the number 2 as it was already rigged and therefore the easiest to animate at this stage. Before I could animate the 3D model, I decided to "purify" it as there was a lot of useless part, such as screw, bolt, cylinder. Doing so I reduced the size of the 3D model file by approximately the half. I used Blender to view and modify the file.

3.2 Animating

Once all the Software, model setup was done I could start to animate the model. To do so I used once more Blender, this time to animate the model we refined. I managed to

do a few animations after a long try out, and suddenly a part of the arm was a few pixels away from the location it should be. This occurred on every animation and even the base refined model. I had to revert all changes done to the model, so all the animations were lost. I looked for another solution rather than continuing with Blender and maybe having the same issue again later. I chose to use Unity's built in animator as a solution. It worked way better than I expected, was faster to develop than with Blender as well. I created a set of three times four different animations which pick up or drop an imaginary object on the ground, in mid-air or high in the air. Each one of those set is composed by one animation which turns to 90° then come back, one which turns to 180° and the two others are the same but in negative (-90°, -180°). I added all 12 animations to an animator which is linked to the Kuka model. This animator is a state machine, it starts in the Idle state. To make it play different animation (state) we change the value of an integer parameter (AnimParam) with different values which correspond to transition reference to one special animation. When an animation is finished playing it sets the value of the parameter to 0, which is the value of the Idle state, so the animator is ready to play another animation.



To play an animation I decided to add a listener which catch if the space key is pressed and then play a random animation using UnityEngine.Random.

```
randomIndex = UnityEngine.Random.Range(0, totalNumberOfAnimation);
```

The second option to play an animation is a more visual one, it allows the user to choose the amount of animations he wants to play. It can be chosen with a slider at runtime and then proceeded by clicking a button bellow the slider. This will add random integer, using the same UnityEngine.Random as before, to a stack. This stack will be used as a "playlist" to play one animation after another. However, I encountered a problem in which the AnimParam wasn't set to 0 after an animation was over. Which leads to not being able to play any other animation or repeating the same one over and over again. So far, the proof led me to think it was a frame problem, so I used a frame counter as a "forced break" after every animation so the AnimParam get properly set to 0. This works, but now sometime the animation queue just stops playing. A solution could be using coroutine instead of the frame counter and have it yield after an animation is done playing so it would not overlap with the starting one.

4 Voxel world

What is a voxel ? For this we need a bit more background about how to represent graphics on a computer. There is two main ways of doing so, vector and raster. "Vector graphics describe the image with mathematical equations, usually representing things such as lines, curves and shapes. Raster graphics instead describe the image as an array of color values that are positioned one after the other into a grid pattern." Pixels are the representation of 2D raster graphics where voxels are the 3D version of it. A voxel is a cube unit that divide a volume in 3D, also called volumetric pixel. It can be imagined or represented as LEGO for more convenience.

The idea was to create a voxel world where boxes susceptible to be occupied by the robot in a close future would change to show this possibility.

4.1 Set up

The first step was to create our own voxel world. It is based on a plane used as ground, as the plane is square its x and z values are the same. The height is calculated every time a call to create the layout is made, it depends on the size of the boxes used as voxel unit. The bigger the boxes are, the less floor the world will have and conversely. The size of the boxes is calculated whenever we want to change the layout, either at the start or when we change the slider and press the button. It simply divides the width with the number of box we want on one side. The position of the first box is calculated as follow.

```
private void CalculateStartPosition() {
      float offset = sizeOfBox / 2;
      float startPos = -(widthLength / 2);
      posX = startPos + offset;
      posY = offset;
      posZ = startPos + offset;
  }
```

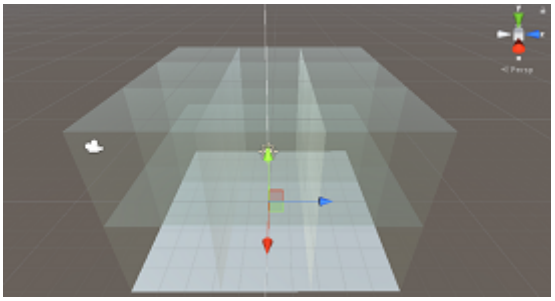
This gives us the position on the edge at -x, -z on the "first floor". To get the next position we just add the size of the box to the position vector at the x component. We control that the position is still on our ground and accordingly start reset the right component and increment the next one. Here is the algorithm which fill the world.

```
private void InstantiateBoxes(int nbrOfBoxesOneSide,
                              GameObject boxToInstatiate,
                              Vector3 position) {
      DestroyPreviousBoxes();

      // Instantiate the boxes with a triple for, one for each dimension.
      for (int y = 0; y < height; y++)
      {
```

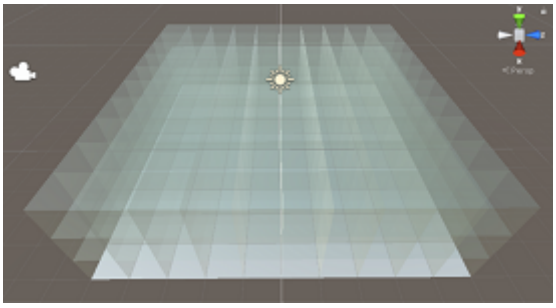
```
        for (int z = 0; z < nbrOfBoxesOneSide; z++)
        {
            // Instantiate one box after the other
            for (int x = 0; x < nbrOfBoxesOneSide; x++)
            {
                // The last parameter link this object to a parent,
                // which helps tracking them to destroy them
                boxToInstantiate = Instantiate(box, position,
                                    rotation, boxParent.transform);
                boxToInstantiate.transform.localScale = GetSizeOfBox();

                // The next position to instantiate a box
                position += new Vector3(sizeOfBox, 0, 0);
            }
            // One row after the other
            position += new Vector3(-widthLength, 0, sizeOfBox);
        }
        // One floor after the other
        position += new Vector3(0, sizeOfBox, -widthLength);
    }
}
```

Here are 3 screenshots of the world with respectively 3, 10 and 30 boxes per sides.

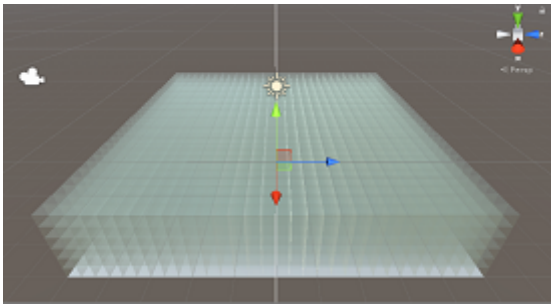
3



10



30

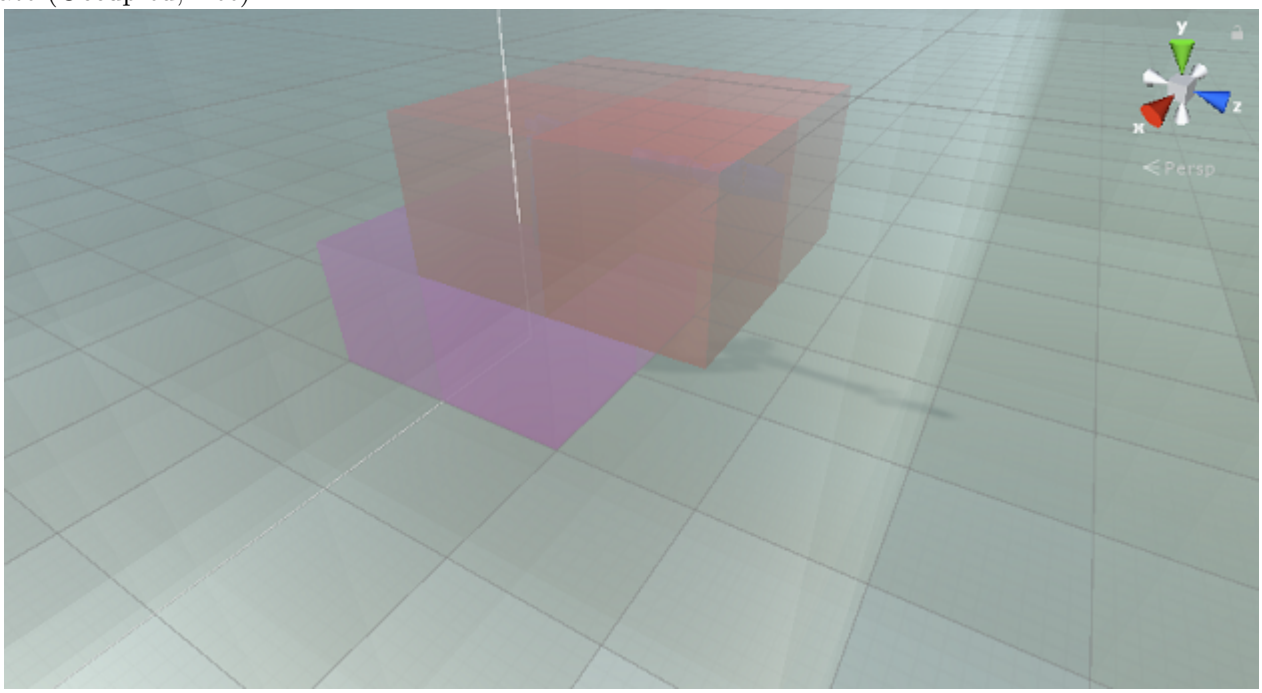


4.2 Collision

As the world was created and could change its density at runtime the next step was to find a way to get the boxes were the robot is. To do so with Unity is quite simple, we needed to add two components to each main part of the robot (here we have 4 main parts). A Rigidbody and a Collider. Those two components had to be added as well to the box prefab which is the gameobject instantiated as the voxel unit. This will let us use the already build-in collision system from Unity. We then use this system with a script called Collider Detector, which is added as a component to the four main parts as well. It will listen to triggers emitted by the collider and called method based on this trigger. There are are three methods used with very similar behaviour, which is changing the material of the collided game object if it has the tag “Box”, the three methods are respectively called: OnTriggerEnter, OnTriggerExit and OnTriggerStay.

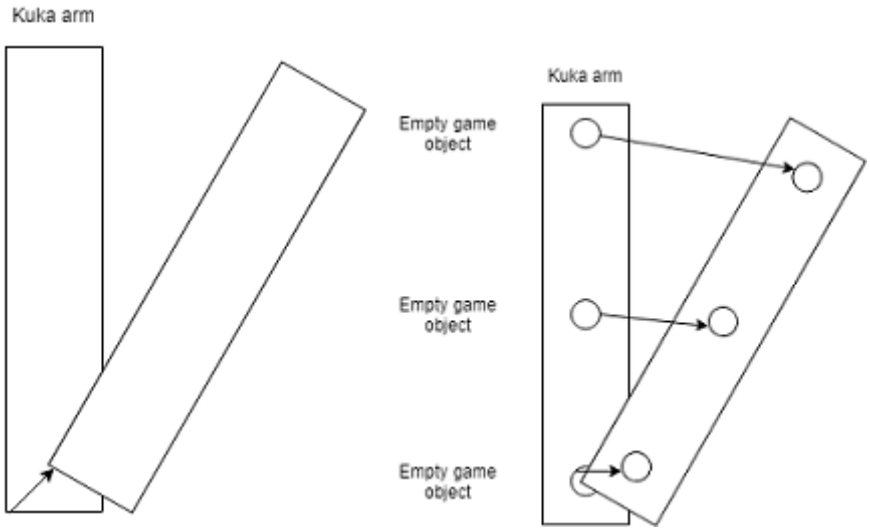
```
private void OnTriggerEnter(Collider other)
  {
      if (other.gameObject.CompareTag("Box"))
          other.GetComponent<Renderer>().material = occupiedMaterial;
  }
```

The tag is used to make sure not to interact with anything other than what we want, in this case the box. Now the boxes change material accordingly to represent their actual state (Occupied, free).

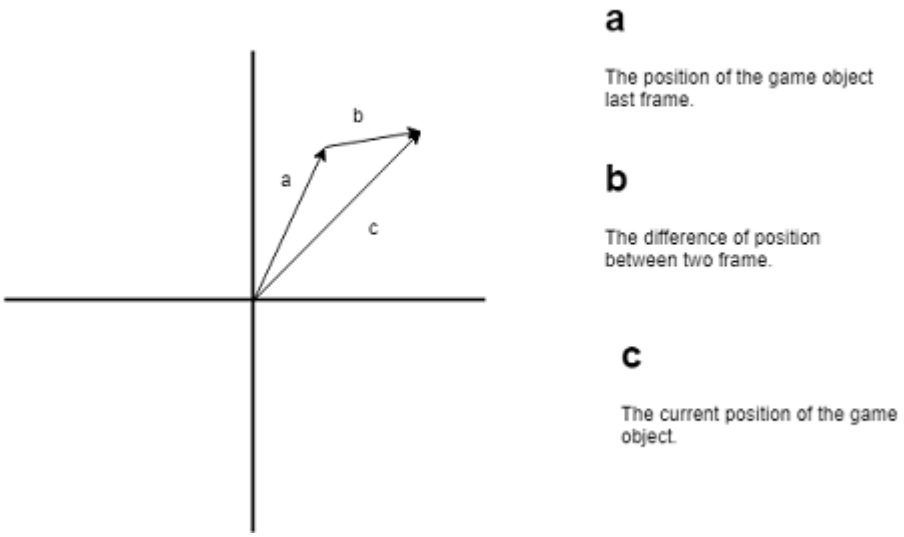


4.3 Prediction

To predict the position of the robot we are using vector addition, but to do so and have decent result we first had to change again our model. Because having only four places where we would calculate the position isn't accurate. We added empty game object to the four main parts and placed them at different places. Here is an example of the forearm to understand the problematic.



One can see that having only one position would lead to very imprecise results whereas it already is more precise with just three helpers. We could then start to calculate with the position helpers, we used vector addition to find the "b" of next figure.



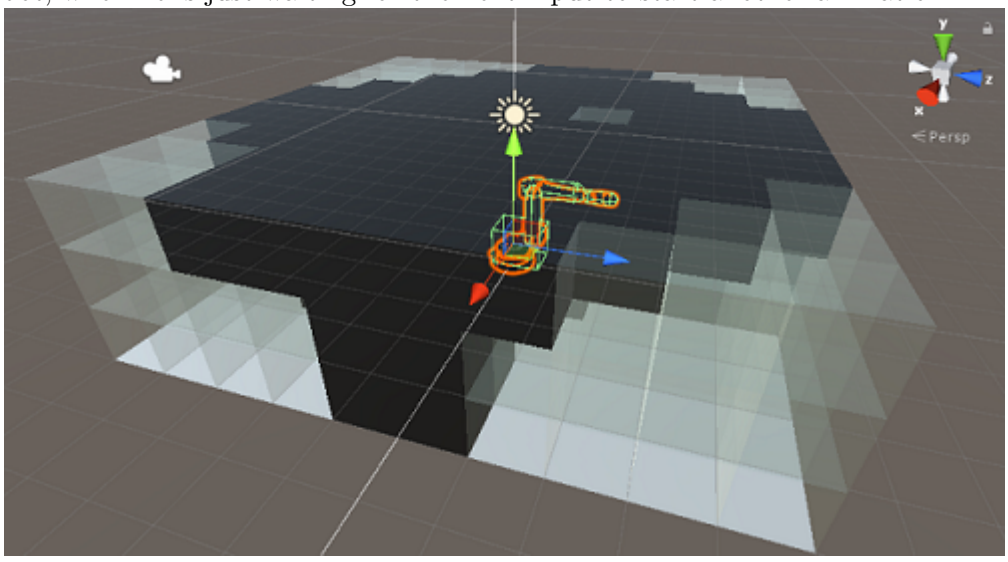
We draw a ray cast from the current position to which we add the vector "b" multiplied with a modifier to give more amplitude to it. Every game object hit by this ray cast will be stored into an array and later checked if they are tagged with the "Box" tag. If it's the case their material will change to show the robot is awaited to occupy this voxel.

```
RaycastHit[] hits;
        hits = Physics.RaycastAll(
          listPositionHelper[i].transform.position,
          listPositionHelper[i].transform.position +
```

```
        (listPositionHelper[i].transform.position -
          positionHelperOldArray[i])
        * vectorModifier,
    maxRaycastDistance);

for (int j = 0; j < hits.Length; j++)
{
    if (hits[j].collider.gameObject.CompareTag("Box"))
    {
        hitList.Add(new RaycastHitBox(hits[j], frameCounter));
        hits[j].collider.
        GetComponent<Renderer>().material = rayCastMaterial;
    }

}
```

4.4 Cleaning

Being able to predict where could be the robot is a nice feature but useless if we don't have a way to notify the voxels that the robot is no longer awaited at their position. Otherwise after a few movements the whole world is filled with possible position of the robot, when he is just waiting for the next input to start another animation.



To keep track of the boxes visited we added them in a list with a timestamp (the frame at which they were hit by the prediction ray cast) just before changing their material.

```
hitList.Add(new RaycastHitBox(hits[j], frameCounter));
```

Every frame we then loop through the list and compare the current frame counter

with theirs, and if it exceeds a given prediction duration we change its material back to the base one and remove it from the list.

```
// Clear boxes which are not expected to be visited anymore by the robot.
private void CleanUnusedBoxes()
{
    // Loop through a list where all collided game objects are
    foreach(RaycastHitBox item in hitList.ToArray())
    {
        // Based on a duration check
        if (item.frame < (frameCounter - predictionDuration))
        {
            item.hit.collider.
            GetComponent<Renderer>().material = baseMaterial;
            hitList.Remove(item);
        }
    }
}
```

5 Conclusion and future work

The visualization of a Kuka robot's working space is a rather bride topic in which I contributed to the first step, where only simulated elements are taking care of.

The stated goals have been reached. The animation of a Kuka robot was achieved with the built-in animation system from Unity3D. It was interesting to reproduce the movement a real Kuka robot would have but the speed of animations is too high to easily see the prediction. Even with the possibility to change the speed of the animation at runtime it is not optimal. Because of the way the prediction is calculated, based on the frame per second, and how the change in speed is changing the rate at which the time in the application goes by, it makes the prediction inaccurate. Predicting the movement of the Kuka robot has been done by calculating a 3-dimensional vector which represent the difference in position between two frames for a given "position helper" and repeating the operation for each "position helper". This 3-dimensional vector is then added to its corresponding "position helper" to represent the position of this part of the Kuka robot in the next frame if its displacement were constants, linear motion. Having the robot being able to only move its arm without himself moving, the base stays at the same position, an improvement in the accuracy of the prediction would be using circular motion and the angular velocity to not calculate a straight line from the "position helper" but an arc. As for now the cleaning of used boxes is arbitrary, a fter a given time the boxes turn back to being "free". And this is done without testing if the robot is still awaited there, it can lead to some gruesome errors. The software is non-suggestible, for human, as its purpose is to predict the position of a Kuka robot and not use a predefined set of case. The environment of the robot can be altered at runtime. The user can create a sequence of animation to be played, enable the prediction or not, change the speed at which the animation is played and rebuild the voxel world with a different number of boxes to gain precision. Having a greater number of boxes slows down the application as each box must control if there is a collision with anything, to gain performance the collision detection should not be done on the boxes but on the robot. Changing the number of boxes populating the world takes some time the software is destroying every boxes object and then creating new ones, if we want forty boxes on one side we have 40*40*6 = 9'600 (width*length*height) boxes in total which leads to a high calculation time. A solution to reduce the calculation time when rebuilding the voxel world would be to create an object pool at the start of the application and store the maximum number of boxes needed in this pool.

The next steps in this project would be improving the current software with said solutions. Using a real Kuka robot instead of a 3D model, capturing its motion with sensors attached to it or with video recognition in real time. And expanding it with Augmented Reality to be able to see with glasses, or another haptic device, in real time the motions of a Kuka robot.