TableCity Documentation

Bien, Seongjin
Neth, Jeremias
Bergenroth, Hannah

February 2020

1 Introduction

TableCity is an augmented reality city-building simulation game, developed for the HTC VIVE Pro on Unity. The gameplay is primarily inspired from tile-based city-building games, such as the early SimCity games. As such, the player is tasked with growing the city, while managing the various needs of its residents. The full detail of the game's logic is described under the Logic section.

TableCity uses the two external cameras in VIVE Pro to project the outside world onto the headset's internal lenses. Multi-marker tracking is used to position the gamespace, which can then be interacted with the VIVE Pro controllers. More details on the tracking component is described under the Tracking section.

The visual assets for this game were created by the group members on Blender, a free-to-use 3D computer graphics creation tool, and Piskell, a simple pixel art creator. The process is discussed under the Visuals section.

2 Tracking

2.1 Development

During development, the tracking underwent several changes. First, we started with *ubitrack*[1]. Due to issues with integrating the software with Unity, we eventually switched to an OpenCV-based ArUco implementation. After that, we combined the inside-out marker tracking with the outside-in tracking of the VIVE's base stations.

Ubitrack This tool offers a multitude of tracking capabilities, one of those is *multi-marker tracking* which we had used in the earlier development stages. While the tracking was sufficient, lack of native integration support into Unity meant the software was highly unstable, leading to frequent crashes. This made development and deployment rather difficult and we had to choose another approach.

[1] http://campar.in.tum.de/UbiTrack/WebHome

ArUco The solution to this problem came in the form of the marker detection functionalities of OpenCV which are based on the ArUco library[2] developed by Garrido et al.[1]. To integrate OpenCV into Unity we rely on an already existing plugin[3]. However, this plugin does not provide all functionalities of OpenCV. One of those is *estimatePoseSingleMarkers* which is essential for marker pose estimation. Consequently, we implemented the function ourselves by using the iterative approach to solving the Perspective-n-Point problem, provided by OpenCV. Furthermore, we've added multi-marker capabilities which relies on user registration but -subjectively- improves the tracking quality and user experience by averaging the estimated poses of multiple markers. Thus, this approach gives us the pose of the markers relative to the camera.

HTC VIVE Pro Integration As previously mentioned, multi-marker tracking gives us the positions and orientations of the markers relative to the camera. Usually, this information is sufficient for applications which solely rely on marker tracking. However, we utilize the HTC VIVE Pro's tracking, of the *head mounted display* (HMD) and its controllers, which is outside-in based by tracking those poses relative to the lighthouses. Here, those send out infrared pulses and the tracked objects effectively track themselves by measuring the time differences at different fixed photodiodes on the object. The integration of the HTC VIVE Pro into Unity is helped by VIVE SRWorks[4]. Now, since the camera is part of the HMD we know the position of the camera relative to the lighthouses. Hence, we obtain the position of the markers relative to the lighthouses, i.e. in world space, by subtracting the camera's position of the markers' positions. Figure 2 displays all of these relations in a spatial relationship graph (SRG), as introduced by Pustka et al. [2]. Figure 1 schematically shows the setup.

2.2 Final Setup

In the game, the main tracking is done by the VIVE Pro. Here, the headset and two controllers determine their positions within the real environment using outside-in tracking. Simultaneously, the headset's external camera constantly scans the environment for any binary square fiducial markers that matches the configured identifiers, and when detected, uses inside-out tracking to position the gamespace relative to those markers. Here, the marker tracking is limited by the camera's relatively low resolution of 640 by 480. This means that the marker detection can easily fail at a distance of 1m or greater, and subsequently the tracking is lost. However, since we know the position of the markers in

[2]http://www.uco.es/investiga/grupos/ava/node/26
[3]https://assetstore.unity.com/packages/tools/integration/opencv-plus-unity-85928
[4]https://developer.vive.com/resources/knowledgebase/intro-vive-srworks-sdk/

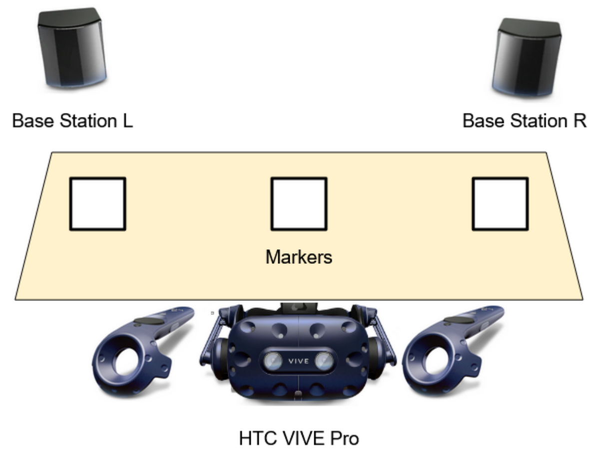


Figure 1: Setup for TableCity

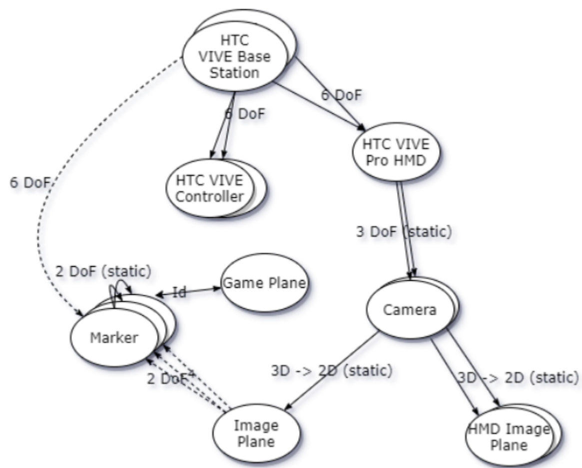


Figure 2: Spatial Relationship Graph of the Setup

world space and the outside-in tracking is still working, we can effectively use the previous tracking as a registration of the gamespace.

3 Logic

3.1 Object Classes

3.1.1 Tiles

TableCity uses a tile system to manage objects in the game. Tile is the fundamental element of the game. Each tile has the following properties:

```
public class Tile {
    public enum Tiletype { Empty, Floor, Road, Residential,
        Industrial, Entertainment, Water, Electricity};
    public TileType Type;
    public Furniture furniture;
    public World world;
    public int X;
    public int Y;
    public bool electricity;
    public bool water;
    public int happiness;
}
```

`TileType` is an enum variable that indicates the current type of the tile, i.e. whether the tile is empty, has a residential/industrial/entertainment building, and so on. Furniture refers to the building currently placed on that tile. Since tiles are managed by the `World` class, which handles tile generation and initial assignment, a reference to it is also included. X and Y indicates the tile's coordinates in the gamespace. The `electricity` and `water` bool variables indicate whether the tile has access to those resources or not. `happiness` is an int variable activated only for residential-type tiles to simulate the number of people living on that tile, and increases based on the types and proximity of nearby tiles. More about them is described under the Resources section.

3.1.2 World

`World` creates the gamespace given its height and width, and initializes a corresponding number of tiles and their coordinates. It also handles calling the correct functions for placing buildings on the tile.

3.1.3 PlayerStats

`PlayerStats` stores the amount of money the player has, and keeps count of the number of each structure the player has built, which is used to calculate the income the player receives in the game.

3.2 Buildings

Different building types were created in *Blender* for each of the `TileType` enum values: `residential`, `industrial`, `entertainment`, `power`, and `water`. The

buildings have different properties and affect the game differently, in terms of incomes, expenses, and resources. The buildings are dependent on water and electricity supply to function as well as being placed next to a road object. `BuildManager` class provides functions for updating the respective tile's properties as well as placing the appropriate meshes of buildings to represent them in the gamespace.

3.2.1 Residential

Residential buildings activate the happiness and population variables in the tile, and consume water and electricity. They serve as the place where citizens of TableCity reside in, and provide workforce and customers to nearby industrial and entertainment buildings. More details on how those resources interact are described in the Resources subsection.

3.2.2 Industrial

Industrial buildings consume water and electricity, and provide a large amount of income to the player as well as a moderate amount of happiness to nearby residential buildings.

3.2.3 Entertainment

Entertainment buildings consume water and electricity, and provide a moderate amount of income to the player as well as a large amount of happiness to nearby residential buildings.

3.2.4 Water and Electricity

Water and electricity buildings, when constructed, activate the electricity and water boolean variables of nearby tiles to indicate that those tiles are being supplied with those resources. Each resource building has a maximum capacity of how much resource it can provide to nearby buildings, and once this capacity is exceeded, additional resource buildings need to be constructed in order to keep up with the resource demand.

3.2.5 Generation of Buildings

To get a greater variety of buildings of the same type, a pool of buildings for each type was created (one for each of residential, industrial and entertainment). From the beginning the opportunity to upgrade the individual buildings and therefore achieving buildings with different properties (e.g. increased population and taxation), was desired. Currently all buildings belonging to the same pool have the same characteristics among themselves but contributes to the visual since it gives the city a better impression when there is a greater variety of buildings represented.

When a building is instantiated on a tile, it is chosen at random from the array containing all prefabs corresponding to the selected building type. It is centered on that tile.

3.3 Controllers

There are three controllers that manage the interaction of various objects within the game.

- WorldController
- ResourceController
- UIController

This subsection will describe in detail the functions of WorldController and ResourceController. UIController will be described in the User Interaction subsection, as it is much more complex and has many more subparts.

3.3.1 WorldController

`WorldController` is the primary controller of TableCity, which manages all other components and interactions between the game objects and the user. `WorldController` initializes the game when it starts, and handles the changes to the respective tile when something is built on top of it, e.g. `water`, `electricity`, `happiness`, and so on. This is managed by `UpdateTileResources` and `UpdateTileHappiness`, which updates the tiles within an n-by-n square with the newly changed tile at the center. `WorldController` also manages the sprite of the individual tiles and road creation, which uses a type of Dijkstra's algorithm to find the shortest path from the origin tile to the tile that the user is hovering over.

3.3.2 ResourceController

`ResourceController` is a simple script that reads the number of tiles with buildings that have their power and electricity requirements fulfilled as well as the happiness of the residential tiles, and grants money to the user every 5 seconds depending on those factors. The equation is:

$$\begin{aligned} PlayerIncome = &10 * EntertainmentBuildings + 15 * IndustrialBuildings \\ &+ ((8 * HappinessRatio) * ResidentialCount) \\ &- 15 * ElectricityBuildings - 10 * WaterBuildings \end{aligned}$$

Note that having electricity and water structures decrease the player's income, while the other structures increase it. This was done to simulate the cost of maintaining resource-generating structures such as power plants and water towers, and the tax raised from businesses and households.

3.4 Resources

In TableCity, the player needs to manage 5 different types of resources, namely: water, electricity, population, happiness, and money.

3.4.1 Water and Electricity

Water and electricity are generated by placing water towers and power plants on tiles. The details about resource propagation is discussed under the Buildings section. All non-empty and non-electricity tiles require a power plant to be built nearby. All non-empty non-electricity non-water tiles require a nearby water tower. Not fulfilling either of those requirements will disable the tile from being counted towards the income generation equation.

3.4.2 Population and Happiness

Population in TableCity can be increased by creating residential buildings. The rate at which population increases is directly related to the happiness of the building, which is in turn related to the residential tile's proximity to other types of buildings. Industrial and entertainment tiles generates more happiness the closer it is to the residential tile, and power plant decreases more happiness the closer it is. The proximity check for happiness is also dependent on those buildings being operational, i.e. having water and electricity supplies.

Entertainment and industrial buildings require residential buildings to be nearby to begin generating income for the player. This is a way to simulate those buildings having customers or employees to carry out their financial activities. An analogy would be a shop (entertainment building) being unable to make sales if there are no customers, and thus will not generate taxes.

3.4.3 Money

Every structure in TableCity costs money to build. Residential, industrial, and entertainment buildings generate income for the player as described in the ResourceController section, and resources structures such as water tower and power plant require money to continue operating.

3.4.4 Resource Indicators

The game provides visual feedback to the users if a building does not have any of the requirements described above. Lack of electricity can be seen by an animated lightning mesh. Likewise, lack of customers/employees or water can be seen by an animated red person-shaped mesh, or a water droplet. Likewise, population and happiness of a residential tile are indicated by the size of the rotating heart and green person-shaped mesh above it. Current amount of money the player possesses is shown within the gamespace, near the (0, 0) corner of the tiles. These indicators are shown in the figures below.

Figure 3: Resource indicators as seen in-game

3.5 Road Generation

The start and end point of the road is determined by the user and is displayed by highlighting the tiles from the selected starting tile to the end tile. This is done to show the user in advance where the road will be placed and before any money will be taken from the user. After the route has been determined, all tiles affected and initially highlighted are stored in a list. For each tile in that list, a road object is created and will be placed on top of that tile in the grid. To get a dynamically generated road structure that changes according to adjacent road objects, a sprite of multiple elements has been created and is used for this purpose. Since a tile can have a northern, eastern, southern or western neighbor, as well as a combination of the mentioned, the sprite of elements is including all scenarios.

3.5.1 Sprite Generation

The generation of the corresponding sprite elements is done by checking the neighboring tiles. For each new road object that is created, it will check whether that tile has any adjacent road objects and assign a sprite to it after that. For instance, if a tile would have neighbors in all directions (north, east, south and west), it will be assigned the sprite element with the name *Road_NESW*. The sprite of elements has therefore taken all scenarios into account, and they are named accordingly for clarity. In the case of a newly created road object that has adjacent road objects, their sprites must also be updated. This is done by firing the callback function of the affected tiles to let it know it has changed and that it needs to update its sprite element.

3.5.2 Restrictions

Restrictions were made where the user can instantiate a `GameObject`. Since it makes no sense to place a building in the middle of nowhere, it must be placed so that it is connected to a road object. This check is performed by `AdjacencyCheck` function in the `Tile` class. Only when it is connected in that way is it reasonable to believe that the building can fully function. A check is made before a building is placed whether there is an adjacent road object

to that tile. Another check that checks that the tile is not already occupied by another object is made, it would be unfortunate to be able to place several objects on top of each other. If it does not satisfy the requirements, the check fails and the user gets feedback that the building could not be placed.

3.6 User Interaction

User interactions (UI) in TableCity are handled by Unity's event system, which detects inputs from the VIVE Pro controllers and provide appropriate visual feedback to indicate its activation. The button mapping of the controllers is shown in the figure below.

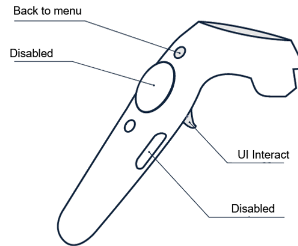


Figure 4: Button functions on the HTC VIVE Pro controller

The player can interact with the tiles in the gamespace by manipulating the controllers, which uses a combination of graphics and physics raycasters originating from the controller's forward transform. The physics raycaster can interact with the individual tiles within the gamespace, which is translated from the global coordinates into local coordinates via a series of coordinate system transforms. This transformation allows the building mesh placement to be locally dependent, meaning the location of the gamespace is independent of the global coordinates, an important feature considering the marker-based localization of TableCity. Pulling the trigger button on the controller brings up the `TileMenu` containing the various building options. `TileMenu` is based on Unity's canvas system, and can be interacted with a graphics raycaster. A canvas object in Unity needs to be configured with a camera from which the graphics raycaster originates from. In a game designed for playing on a standard monitor, this problem is trivial to solve, as the user's primary way of interacting with on-screen buttons is the mouse pointer, which is contained in the primary view rendered in the monitor. However, in a VR/AR setting, the mouse pointer is replaced by the controller, which does not align with the player's primary camera. This meant that a secondary camera had to be attached to the forward transform of the controllers, which was then assigned to the canvas for providing the point of origin for the graphics raycaster. Omitting this step causes the

canvas to be dependent on the orientation of the VR headset, meaning the player has to look at the button in the menu to select it.

In addition to the raycasters, the controllers are also equipped with straight lines directed towards the forward transform of the controller. These "lasers" provide the user with a visual feedback of exactly where the controllers are pointing at to streamline their interactions within the game.

The scripts for handling user inputs from the controllers are contained in `interactUIExt` and `VRInputModule`. `VRInputModule` is a helper object that links the button presses detected in `interactUIExt` to Unity's event system for it to be processed by `TileMenu`.

`interactUIExt` handles the following functions. It keeps track of the tile that the controller is currently pointing at, and places a simple circular sprite to indicate it. When the trigger button is pressed, the script provides the coordinates of the tile that the controller was pointing at at the time of the button press to `TileMenu` and renders the menu right above it. Pressing the trigger to bring up the menu then selecting one of the building options trigger a switch operation, which executes the building function for the respective structure, handled by the `BuildManager`. Selecting the road button instead begins highlighting the tiles between the selected tile and the tile the controller is pointing at using Dijkstra's algorithm, indicating the road that will be built when the player presses the trigger again.

4 Visuals

The different buildings and resource models that are used in the application and displayed in the 3D environment were created in Blender and exported to Unity as FBX files. Template prefabs were used to position and scale the models correctly as well as for the purpose of creating new prefab instances in the scene. Different sprites for the various orientations of the roads were created in Piskell. The result is shown below.

Initially, we considered the option of downloading freely-available mesh packs for the buildings. However, we soon found out that such assets were not available in sufficient quantity, and the ones that could be obtained were not visually consistent with one another.

For presenting gameplay information to the player, we wanted to fully utilize the AR nature of the game, and hence decided to forego a heads-up display (HUD) for 3D displays of the said resources within the gamespace directly. While this sacrifices the accessibility of gameplay information, it enhances the user's immersion when playing the game.

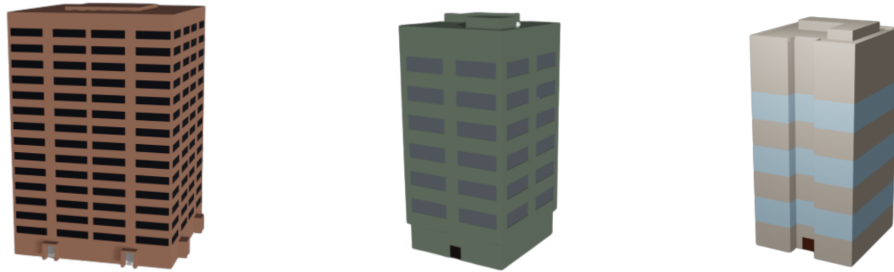


Figure 5: High-rise buildings, apartment block and office

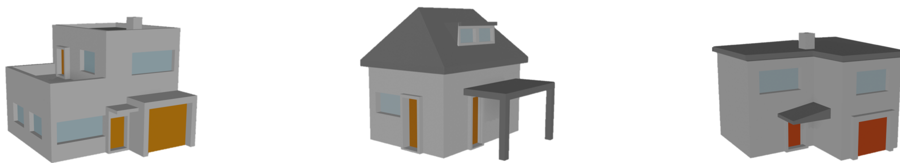
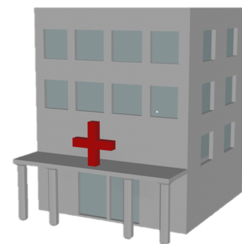


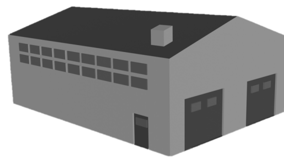
Figure 6: Single family homes

(a) Grocery store

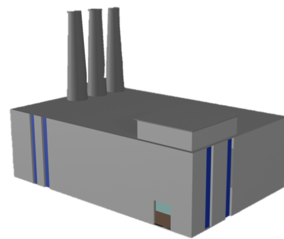


(b) Hospital

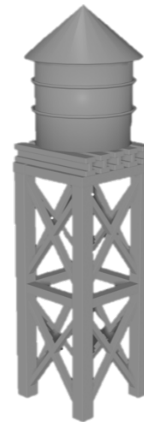
Figure 7: Entertainment buildings



(a) Industrial building



(c) Power plant



(b) Water tower

Figure 8: Resource buildings and industrial building

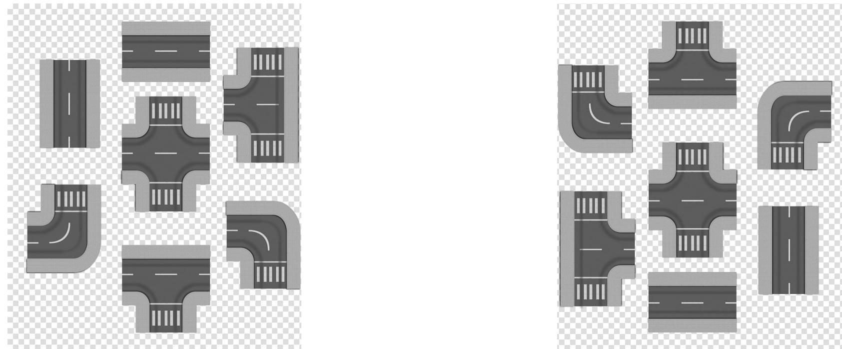


Figure 9: Road sprites

5 Feedback

TableCity was presented at Demo Day 2020, where the team had the chance to present the current state of the game to students and visitors of the event and obtain feedback on the game. The summarized feedback is shown in the figure

below. The actual feedback can be viewed in the appendix at the end of the document.

Overall, the game was well received. However, the suggested improvements were mainly: improvements to the gameplay, specific goals and some form of instructions or feedback on the gameplay elements and what they exactly do. Furthermore, during the tests we observed that some users, those more familiar with the genre, wished for more complexity as the provided content was limited by the time-frame and the game was not as extensive as initially planed. In contrast, the provided complexity was overwhelming to those who were seemingly new to the genre, because the user interface was not elaborated enough and they needed help.

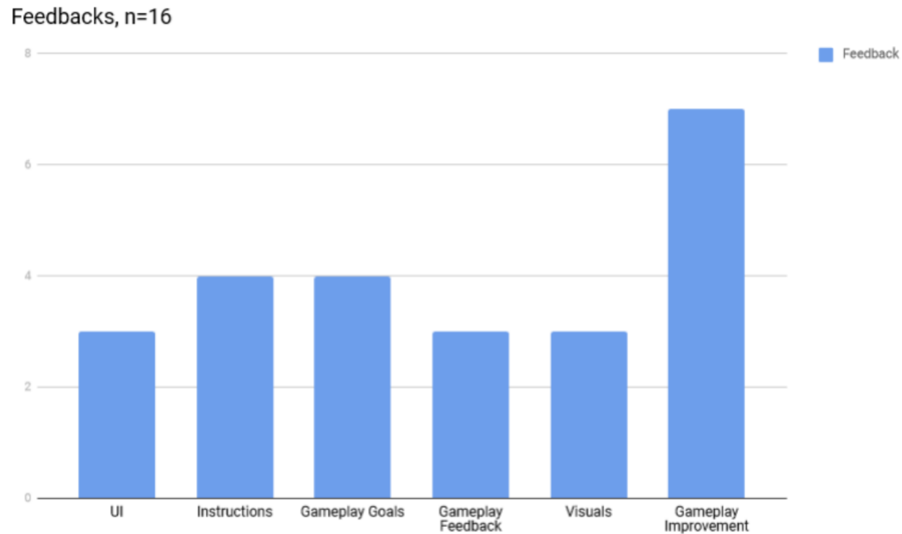


Figure 10: Feedbacks received for TableCity

6 Timeline

The progression of TableCity's development can be summarized in the following way. First, given that setting up the game in AR space would take a long time until it was ready, we decided to develop the game first to be played on a desktop with mouse and keyboard. This allowed us to work on the game's logic and visuals without having access to the hardware and not be dependent on the completion of the AR implementation. Once the desktop version was stable enough, we began porting the game into a VR space, as an intermediate step before the AR integration. There, we were able to make the entire game's

positions dependent on a single local coordinate system, and completed the UI controls with the VIVE Pro controllers. Afterwards, we worked on projecting the gamespace (i.e. the tiles) onto markers seen in a video stream from a webcam. Finally, we integrated SRWorks' camera-to-lens mapping onto the VR headset and worked out the spatial relations of object renderings on the headset's two internal screens.

- Week 1-2:
  - We discussed the fundamental elements of game logic, such as resource types, building types, and the tile system.
  - The basis of the desktop version of the game was developed, and placeholder meshes for buildings were created in Blender.
  - Circle sprite and algorithm for indicating the tile that the mouse pointer is currently hovering over were created and implemented.
  - Multi-marker tracking was implemented using ubitrack.
- Week 3-4:
  - User interaction component (TileMenu) was added. User could now select a tile and click on a button to trigger different functions, which would later be integrated with the building placement algorithm.
  - Road generation algorithm was created, and corresponding sprites for the roads in different orientation were added.
  - More sophisticated meshes for the buildings were created.
- Week 5-6:
  - Road sprite placement algorithm was improved to take adjacent and existing road tiles into account for more consistent visual effect.
  - Building placement algorithm was added.
  - The game was ported into VR space. User interaction was now possible in VR, meaning individual tiles could be interacted with using the controllers.
  - Development on the marker tracking using OpenCV started.
- Week 7-8:
  - Additional assets for the different building types were created, and an algorithm for randomly picking an asset given the building type was added.
  - Road sprites were overhauled to be more visually pleasing. An early version of player money and income were added.
  - Game logic regarding electricity and water propagation was added and fleshed out.

  - Happiness and population interactions were added, and income calculation was finalized.
  - Research into enabling HTC VIVE Pro's external cameras for AR began.
  - Gamespace projection onto the real world image using a webcam showing the markers was completed.
  - After resolving initial issues, including but not limited to confusions with column and row order of the camera matrix and wrong calibration values, with the OpenCV plugin, the marker tracking was finally up and running.

- Week 9-10:
  - All objects within the game were made to be dependent on a single coordinate system. Buildings' transform (size, location, orientation, etc.) was made consistent.
  - Road generation algorithm was overhauled to use Dijkstra's shortest path algorithm for a more organic experience.
  - Spatial relations for rendering the 3D objects onto the two internal lenses were calculated.
  - Game was now at a playable state in AR. Nausea caused by bad camera-to-lens mapping was solved by using SRWorks' built-in mapping shaders. Visual resource indicators were added with animations. Multi-marker support was added to the tracking.

- Week 11-12:
  - A main menu was added, which allows players to view some basic gameplay instructions and change map sizes.
  - TileMenu positioning was overhauled to be consistent with the location of the selected tile, and no longer dependent on the orientation of the VR headset.
  - Background music and sound effects were added. Game logic bugs were addressed, such as income not being calculated properly and building operations at tiles located at the edge of the gamespace.

7 Final Thoughts and Future Work

In conclusion, the provided game puts an interesting twist on the well established city building genre by projecting it into reality. While other games in the genre usually have more features and a higher complexity, this could also be achieved in TableCity with more provided time, as mentioned in the feedback section this was also wished by some users. Furthermore, the marker tracking quality and the see-through abilities of the VIVE are limited by the camera, both would

be resolved by future hardware advancements. Alternatively, another tracking approach on AR city builders could be tried, for example a markerless, camera based approach on a handheld device, such as a smartphone. This would also be more consumer friendly by not relying on expensive, niche hardware.

References

[1] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.

[2] Daniel Pustka, Manuel Huber, Martin Bauer, and Gudrun Klinker. Spatial relationship patterns: Elements of reusable tracking and calibration systems. In *2006 IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 88–97. IEEE, 2006.