P05: Virtual Paintings and Graffitis

Kevin De Keyser, Patrick Eppensteiner, Rasmus Lüscher und Jannik Gartmann

January 2021

1 Introduction

1.1 Motivation

The idea of the project is to create a tool that allows virtual drawings on real walls and objects. The initial project proposal proposed a spray can-like tool to create graffiti. We were really interested in this idea and are motivated to create a tool that makes it possible to create graffiti on real walls without physically altering them. To make this as realistic as possible the focus is on the interaction between the user and the real environment.

There are two main focus points in this project. The first one is the spatial awareness to find the walls in a room and make them paintable. The second one is the painting tool itself.

2 Methods

We used a HoloLens 2 as a mixed reality device for our project. The project uses Unity with C#, the Mixed Reality Tool Kit (MRTK) and the Scene Understanding API.

We changed and improved various features over multiple iterations. We tried various approaches and refined them in the next iteration. We will try to show the major changes in our approach in the next section.

2.1 Painting

2.1.1 Canvas in the Room

Our first prototype consisted of a very simple brush tool and a floating canvas. The canvas was a game object that we placed somewhere in the middle of the room. It is possible to move and resize it using various scripts from the MRTK toolkit. The brush tool consisted of a script that just placed circular game objects onto the canvas whenever the hand ray intersected with the canvas. The hand ray is generated when you pinch the thumb and index finger. This approach causes memory problems because a huge amount of Unity game objects are placed into the scene. The second problem with this approach was, that when moving the hand too fast, individual circles and even spaces in between them were able to be seen instead of a continuous line. Both of these problems lead us to use a Unity Line Renderer. The Unity Line Renderer saves the points where the hand ray intersects the canvas and draws only one game object per line. This solved the continuous line problem since the Line Renderer can just interpolate between the points and had the advantage of a more meaningful history of game objects that can be used for undo/redo functions. The memory also was not a problem anymore with this approach, although still a lot of points had to be saved for an accurate line.

2.1.2 Pinning Textures to the Wall

Using the Line Renderer was only a short term solution for memory problems. As an improvement to the project, we were planning to have multiple canvases, each representing a wall, a different solution had to be found. The solution was a texture based approach. Instead of drawing game objects into the virtual world, the texture of the canvas would be manipulated. This change had multiple advantages. It is fast and efficient, as the texture is saved as an image (or similarly as a texture file) and can now be manipulated as such, which comes with even more advantages like color blending. Color blending was not as simple as with the game object-based approach which would have required to iterate over all game objects, find the spatially close objects and manipulate the appropriate objects to have the desired blending effect. The disadvantages were that all the manipulations had to be coded by ourselves, including the line rendering and undo/redo functions. To improve the realism of the painting, we want to be able to blend the already existing colors with the new color (and the background) to a certain degree. This would be very difficult and tedious if we kept the game object-based approach.

The canvas is now a plane with a texture applied to it. Drawing on the canvas is done by manipulating the texture. This required coordinate transformations since the correct coordinate on the texture (texture coordinates) relative to the ray intersection on the canvas (world coordinates) had to be found. To keep the feature of continuous lines the previous intersection points have to be stored and interpolated using our own implementation of Bresenham's line algorithm. A variable called size is introduced to change the thickness of the line.

Color blending becomes really easy this way by just reading the color of the texture at the point where we want to draw and instead of just overwriting, it is updated with a blending function using the old color value and new color value.

For the prototype version, a feature was added to manually pin the canvas to the closest wall. For this feature, a spatial observer was used from the MRTK to get the mesh of the environment. This was used to pin the canvas onto the mesh using surface magnetism. The goal was that this process is automated and works on surfaces other than walls as well.

2.1.3 The Spray Can Tool

In the third iteration we started experimenting with the spray can tool. As a first approach a spray pattern was created by simulating the paint particles. A number of rays were sampled starting from the tip of the hand that represents the spray can. The directions of the rays were sampled in a cone shape centered around the hand ray. Paint game objects were then generated wherever the rays intersected the canvas. The spray patterns were really good using this approach, but on the other hand, the performance was terrible because ray cast operations are expensive as well as the previous issue of generating and saving many game objects leadsing to memory problems.

In the second approach we modelled the spray can using a mathematical

model, namely conic sections. There are some variables that need to be introduced.

1. r is the hand ray cast from the hand to the canvas
2. p is the point where the hand ray r intersects with the canvas.
3. d is the distance between the origin of r, namely the hand, and p.
4. n is the surface (canvas) normal of the intersection point p.
5. β is the angle between the hand ray r and the surface normal n
6. $\alpha = \frac{\pi}{2} - \beta$ is the angle between the hand ray r and the surface n.
7. ϕ is the rotation between r and the x-axis in local space of the surface. We calculated this by projecting r onto the surface by setting it's local z coordinate to 0. It's used to rotate the ellipse or hyperbola depending on the angle
8. L is the semi-latus rectum, a parameter of a conic section. In the case of the circle it's the radius, in case of an ellipse or hyperbola it's the width of the focal parameter.

Once again we start by ray casting the hand ray to check for a surface interaction. If we find one we calculate p, d, n, α and ϕ. We then use the following formula that defines the conic section

$$r = \frac{l}{1 + e \cdot \cos\theta}$$

e is the eccentricity and l is the semi-latus rectum. In our application we scale l depending on the distance d since we want smaller spray patterns when we are closer to the wall and a larger spray pattern when we are further away. There is a variable called θ that defines the spray angle and can be changed in the user interface. A larger spray angle should result in a larger spray pattern.

$$l = L \cdot \theta$$

If the eccentricity e is 0 the formula will result in a circle with radius l, if it's $0 < e < 1$ then the formula describes an ellipse and for $e > 1$ it's a hyperbola. Therefore we interpolate e depending on the angle α between the surface and the hand ray r. The first fixed point for the interpolation is when the hand ray is perpendicular to the surface. The spray pattern should be a circle and therefore $e = 0$ and $\alpha = 90°$ is the first point. For the second point we look at the angle α where the transition between hyperbola and the ellipse takes place. This is the case when α is equal to the spray angle θ. Therefore the second fix point is $\alpha = \theta$ and $e = 1$. Using linear interpolation we can now get the correct eccentricity e depending on the angle .

We now color a point t on the texture if it satisfies the requirement

$$\sqrt{(t.x - p.x)^2 + (t.y - p.y)^2} \cdot (1 + e \cdot \cos((t.y - p.y, t.x - p.x) + \phi)) \leq l$$

To color a pixel, we sample a random variable that decides if the pixel will get drawn if it's above a certain threshold. When you're close to the wall the spray pattern is smaller but the paint is more dense. It's also easier to draw over already existing color. When you're further away then the spray pattern is larger but less dense. Therefore we change the threshold depending on the distance d. We also do interpolation between the old color that is already on the canvas and the new color that gets sprayed onto it. The weights of the interpolation also change depending on the distance d, meaning that when you're close the new color is stronger and it's easy to draw over old color.

2.2 Spatial Awareness

For the spatial awareness we used the Scene Understanding API after trying various approaches. Enabling it already makes the application to slow, so we had to do various optimizations to make it run-able in real-time. First, for the settings we used mesh-based + coarse and the auto refresh has to be turned off. The scene understanding will still run in the background and generate the meshes/quads. However, the mesh itself is slower than the quad setting. So first, we recommend using the setting quad and then later once the logic is implemented, you have to explicitly remove the mesh generated by the scene understanding whenever the scene understanding API updates. In addition, the scene understanding API detects walls, ceiling, floor and other objects. We only included the settings: Inferred walls, walls, ceiling and floor. This makes it usable for indoor drawing. We were not able to get efficient code running for the scene understanding outside. To draw outside, we recommend placing the canvas manually. Also, we had to reduce the resolution for the auto-generated walls because of the large overhead that the scene understanding API brings.

Note: The scene understanding does not provide coordinates for corners, as the walls/ceiling/floor it generates are not square (they are meshes), so you have to find the corners manually. A trick you can use is use Unity's hitbox API to get the corners easily, this will still give you the exact corners for the walls provided they are square.

For future projects, we would probably assign the canvas directly as a texture to the unity game object created by scene understanding (it creates the walls/ceiling/floor as a mesh) to get an even better performance. Unfortunately, the scene understanding does not create square objects, but instead use general meshes, so our texture based approach might be difficult.

3 Results

The video "Enjoy Yourself: Graffiti with HoloLens 2" https://www.youtube.com/watch?v=VS3SNHswBl4 has been made to showcase what the tool looks like in action. In there you can see various paintings that have been made outside on to real walls. In the figure below you can see one of the example paintings. On the left side there is real graffiti that was already on the wall. We then made a virtual graffiti on the right side for comparison.

Figure 1: real graffiti on the left vs virtual graffiti on the right

3.1 Graffiti

In the figure below you can see the different spray patterns that we can achieve using the spray can. Very narrow angles to the surface make large hyperbola spray patterns. very narrow angles to the surface vector make circles or narrow ellipses.

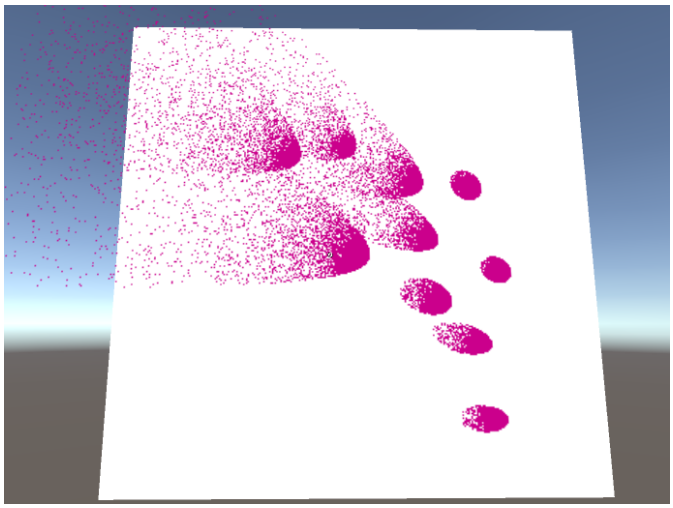

Figure 2: different spray patterns depending on distance and angle

3.2 Spatial Awareness

In the figure below we can see how the spatial awareness placed two canvases on the wall. We can now draw around the edge. The transition from one canvas to the other is almost not noticeable.

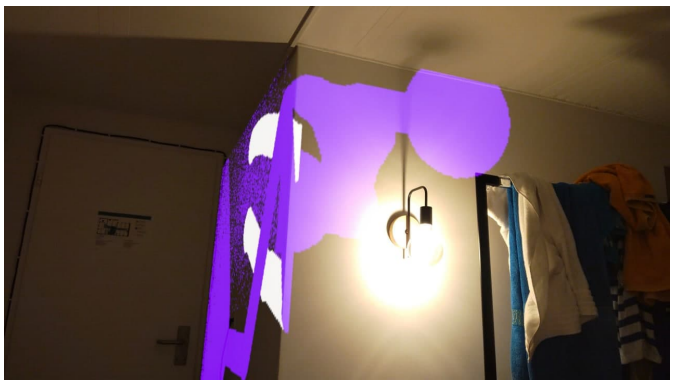

Figure 3: drawing around an edge, detected by scene understanding

3.3 Debug Tool

The debug tool which can be enabled through the menu can display the debug output among other things. Because the debug output is only shown in debug mode in visual studio it is better to use this debug tool, as the debug mode makes the system significantly slower (eye-balled factor 2-16).

Figure 4: the debug menu on the left, console output on the right

In addition, we have various buttons to enable/disable specific settings of the scene understanding API. While we could not get it to run perfectly smooth, thanks to the tool we figured out the optimal settings to run our scene understanding with. It also shows how many walls/ceilings/floors/etc. the scene understanding API recognises. Note this number is limited only by the quality settings and the memory of the HoloLens 2, so it's tricky to get it to function without taking up most of the resources.

4 Conclusion

4.1 Discussion

The scene understanding API used to be in the toolchain of the MRTK, however it was removed and we assume it's due to performance issues. It is still a useful tool, but it only really works indoors and with very coarse objects (like walls). We hope this is just a performance issue, that will get resolved with future iterations of the Hololens.

Also Unity is not the most efficient engine and you have to be very careful to minimize gameobjects (for example creating textures, as we did).

Finally, to get better performance all real-time lighting and shading has to be disabled.

4.2 Future Work

For future projects, we would probably assign the canvas directly as a texture to the unity game object created by scene understanding (it creates the walls/ceiling/floor as a mesh) to get an even better performance. Unfortunately, the scene understanding does not create square objects, but instead use general meshes, so our texture based approach might be difficult.

One way to further improve the tool would be to use spacial anchors to either save canvases and load them at a later point. A second application of spacial anchors that could make this tool more interesting would be too use it them

to synchronise canvases between multiple Hololenses. Either to just view the graffiti together or to actually make collaborative paintings.