# CMP 405 – Tools Programming

1703221

## Introduction

The aim of this project is to extend upon existing functionality of the WOFFC world editor. When designing tools to add to the editor, particular thought is put into how the user will interact with the tool and making it as intuitive as possible.

## Controls

### Camera

All movement controls are relative to the up vector of the camera.

W = Forwards

S  = Backwards

A  = Strafe Left

D  = Strafe Right

E = Up

Q = Down

Shift = Speed Boost

Hold right click to move camera direction.

### Functionality

C = Undo

V = Redo

F = Duplicate

Left click to select objects.

Left click and drag on objects for transforms.

Left click for terrain manipulation.

Scroll wheel to change radius of terrain tool.

## Camera

The camera is encapsulated into its own class. It is a first-person camera that can be manoeuvred in any direction (except roll). The direction of the camera is controlled with the mouse. When the right mouse button is pressed and the mouse is moved, the cursor's x and y offset from the screen centre adjust the yaw and pitch of the camera, respectively. The pitch of the camera is clamped between 90 and -90 degrees to avoid disorientation.

The roll, pitch and yaw of the camera are calculated by using the parametric equations of a sphere. A Forward, Right and Up vector is calculated using these values. The camera can then be manoeuvred along these vectors. A Look At vector is calculated by simply taking the Position vector of the camera and adding the forward vector. The Position, Look At and Up vectors are used to generate a view matrix which is used for projection.

This style of camera is what you would expect when playing a lot of computer games or using similar tools, so would likely be intuitive to the user.

When the right mouse button is not pressed, the user is free to move the mouse around the scene and select and manipulate objects and terrain. It is still possible to strafe with the camera in this mode. This ability to switch between using the mouse for camera movement and scene interaction feels intuitive and makes much better use of the mouse.

## Duplicate

At any point while an object is selected, it can be duplicated by pressing the 'F' key. A copy of the currently selected object is created and added to the most recent scene graph. The display objects are then updated to match the scene objects. Duplicating objects with one button rather than a copy / paste system seems more intuitive and allows fast duplication of objects.

## Undo / Redo

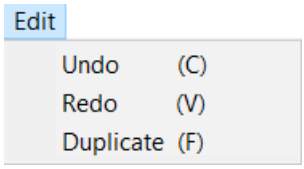
At any point when manipulating objects, you can undo previous transformations. This is achieved by keeping a dynamic array of all the states of the scene objects and keeping track of the current state of the scene. The scene memory can be traversed backwards and forwards (undo / redo). If an object is manipulated when the scene pointer is at a

point in the past, the scene states after that point are deleted and a new branch of scene states is created.

The ability to undo mistakes will be expected as standard by the majority of users. Without this feature, development would be very frustrating.

## Menu

Undo, Redo and Duplicate can all be accessed from the MFC menu bar under Edit. Hotkeys are displayed next to commands for quick access.




## Mouse Picking

Mouse picking is achieved by casting a ray from the mouse position to the object and doing AABB collision with the ray and the bounding box of the object. The ray can intersect with many objects, so the intersected objects are sorted by distance. The object with the shortest intersection distance is the object that is selected.

The most intuitive way to select an object is to click on it. This is the first thing most users will try. It allows for faster interactivity and gives the user more control.

## Model Imports

Basic models were created in Blender and exported in the form of OBJ and MTL files. These were then converted to DDS files using the DirectXMesh command line tool, “Meshconvert”. These models included a low polygon tree and various widgets for user feedback while editing objects and the environment.

## Keyboard / Mouse Input

When detecting keyboard input from Windows messages, as well as having an array to keep track of the currently pressed keys, an array is also kept of the key states of the last frame. This is useful for detecting the initial frame of when a key is pressed or released. A similar approach is used for the mouse, keeping track of the previous mouse position and mouse button states. The previous mouse position is used for calculating the difference in position between frames for manipulating objects.

## Toolbar

The toolbar consists of various buttons to switch modes when using the application. The images are bitmaps, designed to indicate the functionality of the button.

This button is the default state of the application. Objects can be dragged around the map freely by moving the object to the intersection point of the ray cast from the mouse to the terrain.

These buttons are for translating objects along the x, y, and z axes, respectively.

These buttons are for scaling objects along the x, y, and z axes, respectively.

These buttons are for rotating objects around the x, y, and z axes, respectively.

These buttons are for terrain editing.

This button switches to a “Build” tool for raising the terrain in a radius around the mouse position.

This button switches to a “Dig” tool for lowering the terrain in a radius around the mouse position.

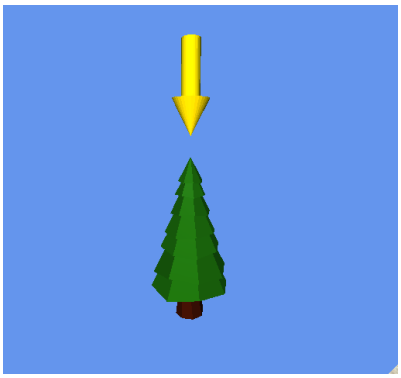
This button switches to a “Flatten” tool for flattening the terrain to the height of the terrain at the mouse position when it is initially pressed, in a radius around the mouse position.

This button switches to a “Smooth” tool for smoothing out the terrain in a radius around the mouse position.

## Widgets

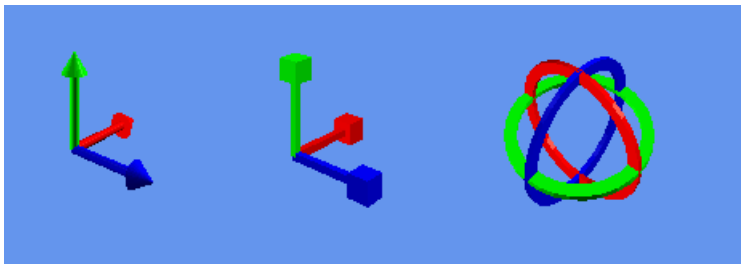
There are various widgets in the application to give feedback to the user. A lot of thought was put into UX design here. The goal was to make the tool as intuitive as possible.

### Selection Widget



### Axis Widgets

When an object is selected, a big yellow arrow is displayed above the object. This arrow moves up and down for added visibility.

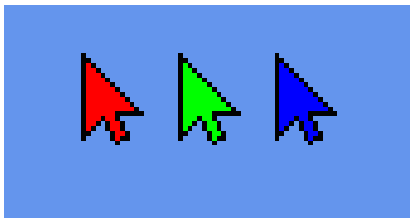


When an object is being transformed in relation to an axis, an axis widget is visible to give the user an indication as to which tool is selected and what orientation they are in. Red, green, and blue correspond to x, y, and z, respectively.

The arrow widget represents translation, the square widget represents scale, and the circular widget represents rotation. In the rotation widget, the circles orbit the axis of rotation.

These widgets are transformed to be just in front of the camera and always facing the correct direction. To achieve this, two matrices are needed, the camera's world transform and the widget's local transform. The widget's local transform multiplied by the camera's world transform gives the desired translation of the widget. The rotation of the widget is not altered to keep the axes pointing in the right direction.

**Coloured Cursor**



When using any of the transformation tools, the mouse cursor changes colour to match the axis the object is being manipulated in relation to. This gives the user visual confirmation about what it is they are trying to do. The colour of the cursor corresponds to the axis on which the object will be transformed. So, at a glance at the axis widget, they can tell what direction to move the mouse in to manipulate the object.

Changing the cursor colour was achieved by making use of some low-level windows functions, which required a lot of research on MSDN. (Using Cursors - Win32 apps, 2021)

CURSORs require an ICON that require two BITMAPs to render the cursor to the screen. BITMAPs are made up of COLORREF objects. These are 32 bit words. They hold the red, green, blue, and alpha components of a colour in the format ABGR. There is an AND bitmap and an XOR bitmap.

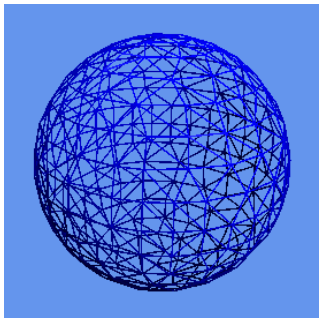
The AND bitmap is a mask. It is used like a stencil. The pixel colours of the screen behind the cursor are combined with the AND bitmap using the bitwise AND operator. Any bits in the AND bitmap that are set to 1 will allow colour from the screen pixels to be displayed (Jiju, 2021). So, in order to achieve transparency, wherever the cursor image should be transparent, all the bits in the AND mask should be set to 1, so the COLORREF should be set to 255 in the red, green, blue, and alpha channels.

After the AND mask bitmap has been applied to the screen pixels, the XOR bitmap is applied using the bitwise XOR operator. The XOR bitmap is essentially the image of the cursor. Wherever the cursor image should be transparent, these bits are set to zero, so the screen pixels will remain unchanged. Wherever the cursor image is opaque, the XOR bitmap is set to the desired colour.

To create the bitmaps, a handle to the device context for the client window is needed. Then two bitmaps and compatible device contexts are created for the AND and XOR bitmaps. For the XOR bitmap, Windows GDI is used to draw a primitive polygon. The AND bitmap is created by looping over each pixel in the XOR bitmap and setting any bits that should be transparent to 1 and any bits that should be opaque to 0.

The screen pixels combined with the AND and XOR bitmaps has the desired effect of creating a custom mouse cursor.

**Terrain Widget**



When using any of the terrain editing tools, a low polygon, wireframe sphere is visible to indicate to the user the radius around the mouse where the terrain will be edited. The size of the sphere can be manipulated by using the scroll wheel of the mouse. This widget always snaps to the closest vertex to the intersection point of the ray and the terrain. This ensures a predictable circular shaped deformation.

## Snapping Objects to Terrain

Unless the user specifically states that an object should be floating above the terrain by using the translate y tool, the objects are consistently snapped to ground level whenever they are translated, or the terrain is edited underneath them.

This is achieved by casting a ray straight down from the maximum height of the environment at the x / z position of the object. The total

height of the environment minus the distance to the ray intersection is the y position of the ground at that point.

This snapping to ground level allows the user to easily place objects on uneven terrain without having to manually edit the y position of the objects.

## Object Transformations

When manipulating an object with respect to a specific axis, a ray is projected onto the plane running along that axis, through the object.

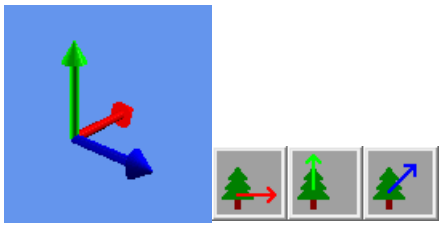
A plane can be described with a normal vector and a point on the plane.

The plane running along the x axis has the normal (0, 0, 1).

The plane running along the z axis has the normal (1, 0, 0).

The point on the plane is the object's position.

### Translation



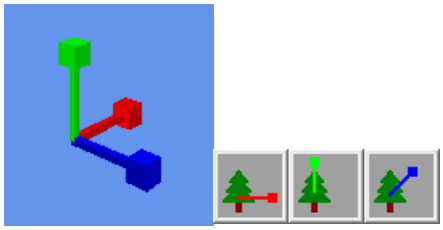
When translating along the x axis, the object's x position is set to the x component of the intersection point of the line and the plane.

When translating along the z axis, the object's z position is set to the z component of the intersection point of the line and the plane.

When translating along the x and z planes, the object's y position is set to the height of the terrain at this x / z position.

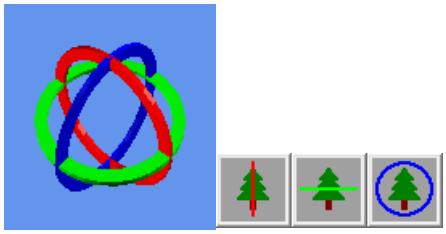
When translating along the y axis, either the x or z plane needs to be used instead of the y plane as the object's y position needs to be set to the y component of the intersection point of the plane. On the y plane, y always equals zero, so this does not work. In order to choose the best plane to use for this purpose, projection is used. The absolute value of the dot product between the camera's forward vector and the plane's normal is calculated. The smaller value is the plane that is facing the camera. This is the plane that is used.

### Scale



When scaling objects, the same planes are used as for translation. The object's scale vector is set to the absolute value of the difference between the intersection point and the object's position on a respective axis. The absolute value is used so that the object model does not invert.

### Rotation



When rotating objects, it felt more intuitive for the user to be able to rotate objects in relation to world space as opposed to local space.

To achieve this, when the mouse is initially pressed on the object, a local rotation matrix is created using the yaw, pitch and roll attributes of the object's rotation vector.

When the mouse is moved along the planes (like in translation and scale), a new parent rotation matrix is created using the DirectX::SimpleMath::Matrix::CreateRotation(X / Y / Z) functions. The angle passed into the function is the difference between the intersection point and the object's position on a respective axis (like in scale).

When rotating on the y axis, the planes are not used. The angle is simply calculated by the change in mouse position along the x axis since the last frame. When dragging the mouse left, the object rotates clockwise. When dragging the mouse right, the object rotates anti-clockwise. This is what would be intuitively expected.

By multiplying the local rotation matrix by the parent rotation matrix, you get the resulting world rotation matrix. From this matrix it is possible to extract the yaw, pitch and roll by using some trigonometry on the raw data of the matrix.

With a matrix in the form:

$$\begin{bmatrix} \_11 & \_12 & \_13 & \_14 \\ \_21 & \_22 & \_23 & \_24 \\ \_31 & \_32 & \_33 & \_34 \\ \_41 & \_42 & \_43 & \_44 \end{bmatrix}$$

Roll = atan2(_21, _22)

Pitch = asin(-_23)

Yaw = atan2(_13, _33)

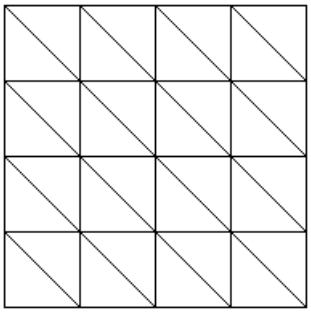
The advantage of transforming the objects using this technique of projecting a ray onto a plane is that the object is always set to the mouse position on the selected axis. This feels intuitive and gives more control to the user.

## Terrain

When checking for intersections between the ray cast from the mouse to the terrain, the simplest approach is to loop over every triangle in the terrain mesh and perform a ray / triangle intersection test on each. This approach is very expensive, however. To solve this, spatial partitioning is utilized. The environment is split into 16 chunks (4 x 4) using DirectX::BoundingBox objects. A singular AABB collision check is now performed on each of the bounding boxes. The triangles that make up each section of the terrain mesh are only checked for collision if there is a collision on the surrounding bounding box. This eliminates the need for the majority of triangles to be tested for collisions and greatly increases performance.

When a collision is registered, the intersection point is stored in a map data structure along with the distance to the intersection. At the end of the collision checks, the map is sorted by distance and the closest intersection is returned.

The terrain is made up of a grid of quads, split into triangles.



The terrain resolution is 128 x 128. That is, there are 128 x 128 vertices in the terrain mesh. There is also a height map that is the same resolution that stores height values for each vertex in the terrain mesh. This is what gets manipulated, and then the terrain is updated from that.

To find the correct indices in the height map to manipulate when editing the terrain, the closest vertex to the intersection point needs to be calculated.

The terrain mesh extends from -256 to 256 on the x and z planes. The heightmap is from 0 to 128. It is possible to find the four corners of the quad that the ray is intersecting with. To find the top left corner we use this equation:

```
Vector2 top_left(((int)(intersection.x + 256)) / 4, ((int)(intersection.z + 256)) / 4);
```

The integer casting clamps the value to the corner. The other 3 points in the quad can be calculated by adding 1 in the required axes.

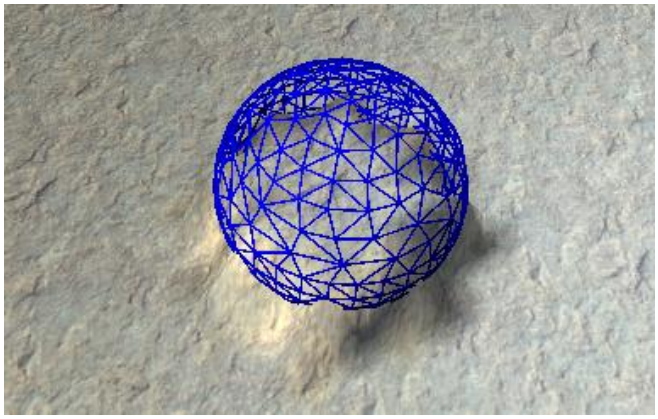
The same equation is used to map the intersection point of the ray, except there is no integer casting involved. This gives a floating point vector between the four corners of the quad.

```
Vector2 intersection_point((intersection.x + 256) / 4, (intersection.z + 256) / 4);
```

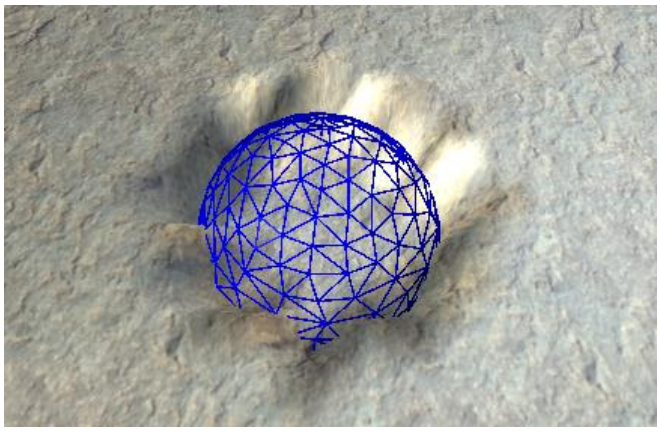
The distances between the mapped intersection point and the four corners of the quad are then calculated. The closest vertex is the vertex with the smallest distance to the intersection point.

Neighbouring height values in the heightmap are manipulated if they fall within a circle around the targeted point. Instead of looping over every point of the height map, only the points in a rectangle around the targeted point are considered. The rectangle’s upper and lower extents are the radius of the terrain tool sphere, clamped to the edges of the 2D height map array.

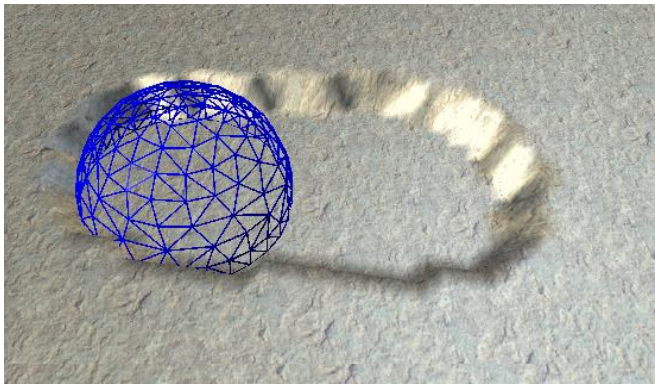
When the terrain is edited, all objects within the radius of the terrain tool sphere are snapped to the height of the terrain.



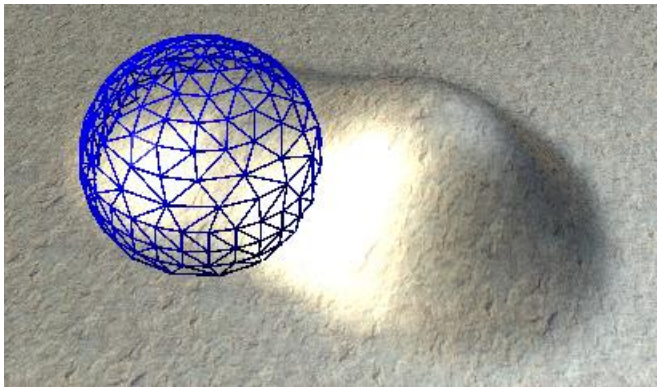
 The Build tool raises the vertices of the terrain inside the area of the sphere.



 The Dig tool lowers the vertices of the terrain inside the area of the sphere.



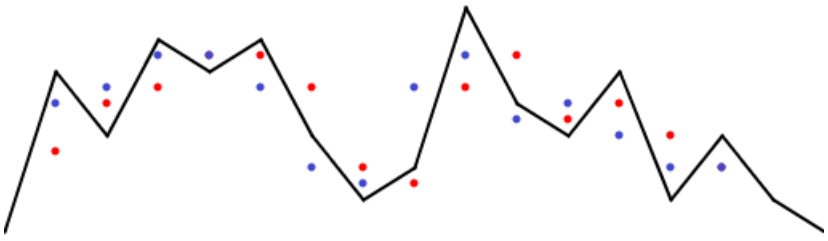
 The Flatten tool sets the position of all vertices of the terrain inside the sphere to the height of the closest vertex to the mouse when the mouse is initially pressed.



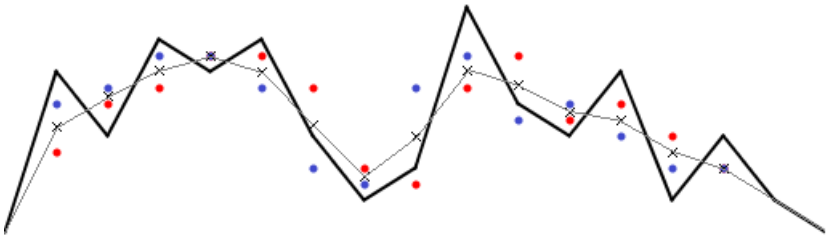
 The Smooth tool is a bit more complicated. It has the effect of smoothing out bumpy terrain.

The algorithm was initially designed on a one-dimensional line.

For each vertex except for the very first and very last vertex:



Calculate the mid-points on the y axis between itself and the left and right vertices.



Then calculate the mid-points between these pairs of points. When these points are joined together, the resulting line is a smoothed version of the original line that still maintains the basic shape of the original jagged line.

Over multiple iterations the line becomes smoother until it is eventually just an arc that sinks to a flat line between the start and end point.

This algorithm expands to two dimensions. Instead of taking the left and right vertex in the one-dimensional case, the North, South, East, and West points are taken on the two-dimensional terrain mesh and the same logic is applied.

## Problems Faced

### Camera

When Attempting to snap the mouse back to the centre of the screen to achieve an unlimited camera rotation with the mouse, I ran into the problem that the function used to return the mouse position returned the position in screen coordinates as opposed to coordinates relative to the client area of the window. I found that I needed to get a windows RECT object that described DirectX child window. Setting the mouse position to the top left corner of this RECT plus the half width and height of the child window resulted in the mouse being in nearly the correct position, however the camera was still rotating slowly to the bottom right. The position I was setting the mouse to was off by 2 pixels in the x and y.

Finding the cause of this offset was tricky to debug. I first corrected the offset manually with a hard coded value, but I was not content with this as I was worried the offset would be different on different computers. I found the problem was caused by the CREATESTRUCT object used to create the DirectX window. Its style was set to WS_EX_CLIENTEDGE.

```
cs.dwExStyle |= WS_EX_CLIENTEDGE;
```

Removing this line of code from MFCRenderFrame.cpp corrected the offset of the mouse position.

### Coloured Cursor

After successfully creating a coloured cursor, I ran into some problems with the cursor flickering when moved. When setting up the DirectX window you can set the default mouse cursor that will be used in the window. After a lot of research I found that MFC intercepts Windows WM_MOUSEMOVE messages and it was setting the cursor to be its default cursor. In ToolMain.cpp I was also intercepting the WM_MOUSEMOVE messages and was setting the cursor to the custom colour cursor. This is what was causing the flickering. Setting the default cursor to NULL when registering the DirectX window caused the flickering to stop.

I was still getting flickering in the toolbar area of the window. My best guess is that similar behaviour was occurring. I was not able to find how to disable the cursor for the toolbar. I got around this by only setting the cursor to be coloured if the mouse y position is greater than the “top” value of the RECT of the DirectX child window.

### Model Imports

When importing models into the project, I could not manage to get models with image textures to load using the DirectX::Model::CreateFromCMO function. The tool we used to create the CMO file, called “meshconvert”, assumes models are created in Maya. I did not have access to this software as the university was closed, so I used Blender instead. Something was not compatible. I got around this problem by changing the style of the models I wanted to import. I created some low polygon models and textured them with default materials. These were just solid colours but with the low polygon models it looked quite good. I was able to generate the CMO files with these generated OBJ and MTL files.

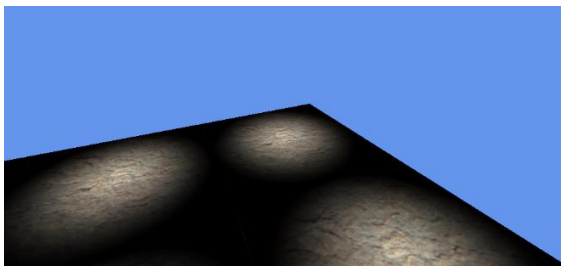
### Object Transforms

When translating objects along the x and z axes, I initially moved the objects by the difference in mouse x position. This technique appeared to work fine initially but when viewing the object from the opposite direction, the object would move in the opposite direction to the mouse.

This makes sense, however it seemed very unintuitive to control the object like this. I tried changing the direction of the translation based on the orientation of the camera, however this technique required a lot of conditional statements and I was not satisfied with the results. I decided to re-think the approach. I wanted the objects to stick to the mouse. As we had just been using ray tracing for mouse picking, this was fresh in my mind. I had the idea of projecting a ray onto the plane along the desired axis and translating the object to that point. This technique proved to be effective at achieving intuitive object manipulation.

**Terrain Painting**

I did attempt to implement terrain painting. I was hoping to achieve gradient based texture lerping, where the terrain texture would lerp between a rock texture and a grass texture depending on the gradient of the geometry. Before that though, I needed to be able to simply lerp from one texture to another. I did some research into the DirectX::DualTextureEffect from the Microsoft DirectXTK Documentation (microsoft/DirectXTK, 2021) and I was able to implement the effect in the example.



“The first texture defines the basic colour of the object, which can then be adjusted by the second texture. Where the second texture contains 50% grey, the colour is unchanged. Where it is darker than 50%, the colour is darkened. Where it is brighter than 50%, the colour becomes brighter. If the second texture is not monochrome, the colour is tinted accordingly.” (Hargreaves, 2021)

I could see how it could be used for light maps, to pre-bake lighting effects at compile time, or apply texture detail over the entire terrain, however I was unsure how to achieve the desired terrain painting effect with this technique. The second texture only adjusts the first texture, it does not replace it. This effect does give you the ability to move the texture co-ordinates for each vertex which could have possibly been used to my advantage. But there was also the problem, that even if I was able to achieve the texture lerping effect, the effect does not support

normals for lighting. And even if I could pre-bake the lighting into the texture somehow, the terrain is dynamic so the normals would need to be updated at runtime. With all these complications, I decided to focus on polishing the tools I had already developed instead of proceeding with terrain painting.

## Conclusion

The tools created in this project give the user the ability to manipulate the transforms of objects quickly and accurately and allow for fast, accurate terrain deformation.

The widgets give constant visual feedback to the user. The coloured mouse cursor combined with the axis widgets make it clear to the user what axis they are manipulating and where the axis is oriented. By manipulating objects in respect to the intersection point on the plane on the respective axis, the objects stay with the mouse which is satisfying. The terrain tool widget, snapping to the closest vertex, is a nice feature as it gives accurate user feedback as to what vertices will be manipulated. The ability to change the size of the sphere provides the opportunity for fine detail, with the sphere shrinking to the point of highlighting a singular vertex.

If I was to attempt a project like this again, I would take more time to familiarise myself the functionality of the libraries available to me.

Before discovering the DirectX::SimpleMath library and all its very convenient, highly optimized maths functions, the ray / triangle collision code was written from scratch using online resources. (Ray Tracing: Rendering a Triangle (Ray-Triangle Intersection: Geometric Solution), 2021)

It works; however, it is about twice as slow as the DirectX equivalent. This code is left in the project and can be used instead of the DirectX equivalent by changing the pre-processor conditional statement from 1 to 0 inside the function. This was not a complete waste of time as it gave me a better understanding of the maths involved in the process but having functionality like this readily available definitely speeds up development.

There are so many features that could be added to this project and I plan to continue working on it in the future.

## References


Docs.microsoft.com. 2021. *Using Cursors - Win32 apps*. [online] Available at: <https://docs.microsoft.com/en-us/windows/win32/menurc/using-cursors> [Accessed 3 May 2021].

Jiju, G., 2021. *Creating a Color Cursor from a Bitmap*. [online] Codeguru.com. Available at: <https://www.codeguru.com/cpp/w-p/win32/cursors/article.php/c4529/Creating-a-Color-Cursor-from-a-Bitmap.htm> [Accessed 3 May 2021].

Scratchapixel.com. 2021. *Ray Tracing: Rendering a Triangle (Ray-Triangle Intersection: Geometric Solution)*. [online] Available at: <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution> [Accessed 3 May 2021].

GitHub. 2021. *microsoft/DirectXTK*. [online] Available at: <https://github.com/microsoft/DirectXTK/wiki/DualTextureEffect> [Accessed 4 May 2021].

Hargreaves, S., 2021. *Detail textures*. [online] Shawnhargreaves.com. Available at: <https://shawnhargreaves.com/blog/detail-textures.html> [Accessed 4 May 2021].