# Assignment 3 – 3D Graphics Programs

## Hailey Martin (11606474)

## Task 1: A 3D Function-Viewing Program

### Code Structure

The code is structured in function.js such that global WebGL variables are defined first, followed by useful variables and constants for calculating the appropriate views. The bulk of the WebGL initialisation code as well as generation of function vertices & mesh elements follows in the init() function. Helper functions are defined subsequently at the bottom of the file: render(), which calculates model-view and projection matrices before sending them to the GPU for rendering; handleKeyDown(), which handles the keyboard-based user interaction; and generateFunction(), which produces the points relevant for the function in question.

### Solution Approach

In general, my solution closely follows the *myhat2* example in structure. I first calculate the points of the function. As the z value depends only on the different values of x and y, I simply loop through and calculate every possible row and column intersection point (0-50) and then calculate the result for the z value. This is used to draw line strips along each row. An element array is used to draw line strips along each column, as well as draw triangle strips along each row to display the function plane.

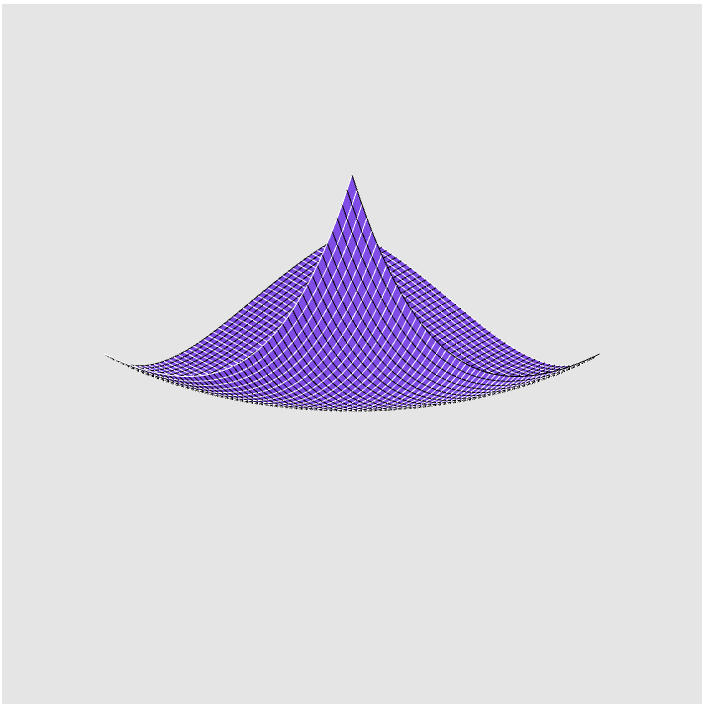
A model-view matrix is calculated with fixed at and up values; up is angled such that the z-axis points up, as required. The projection matrix provides for an orthogonal projection. Polygon offset is used to ensure that the line segments are rendered smoothly above the surface of the function plane, and hidden surface removal is used to block further sections from view.

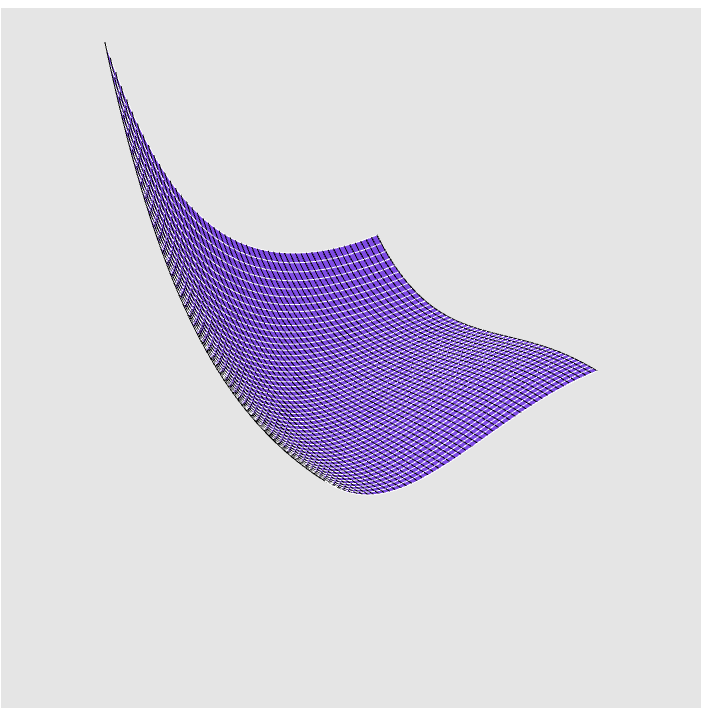
User interaction is handled by handleKeyDown(event) which reacts to keyboard input by adjusting the viewing angle (specifically, the eye value is adjusted by small amounts).

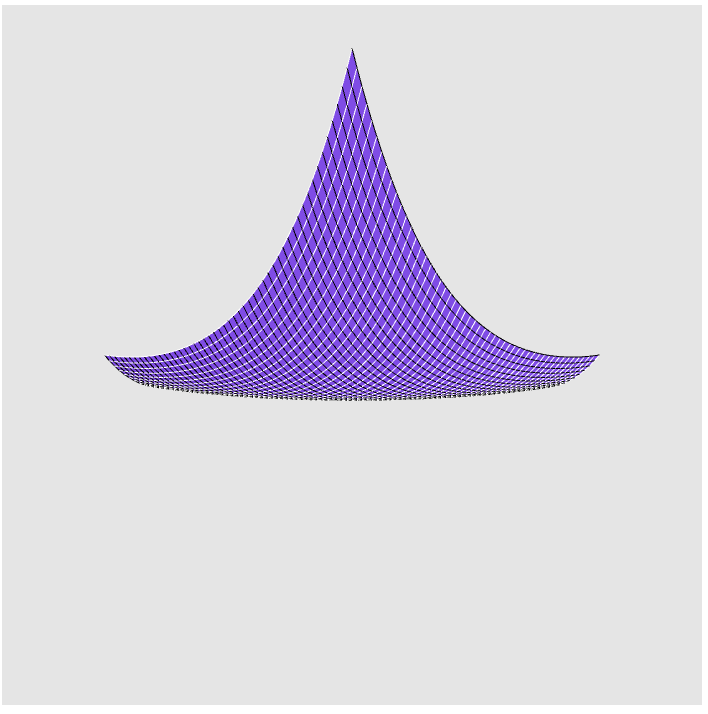
### User Interface

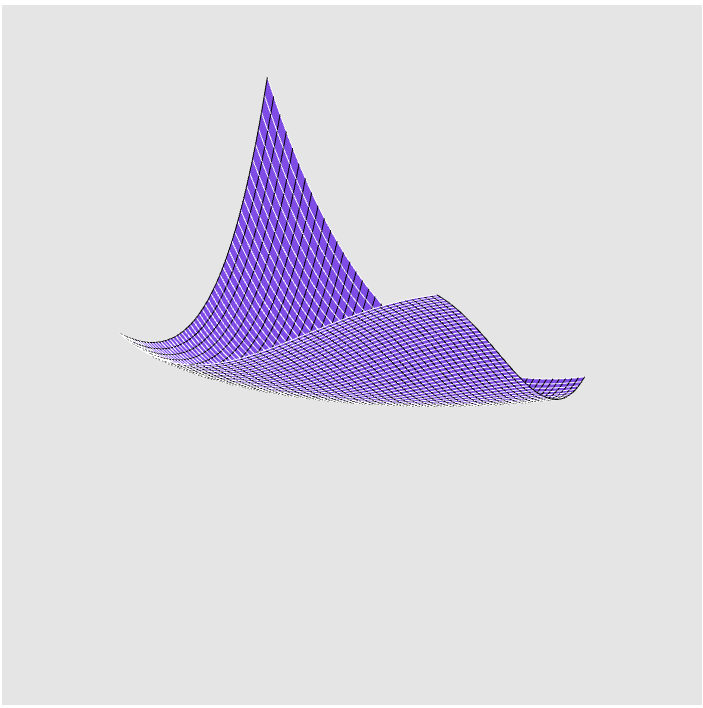
**W, S:** Change viewing position around the north-to-south axis  
**A, D:** Change viewing position around the east-to-west axis

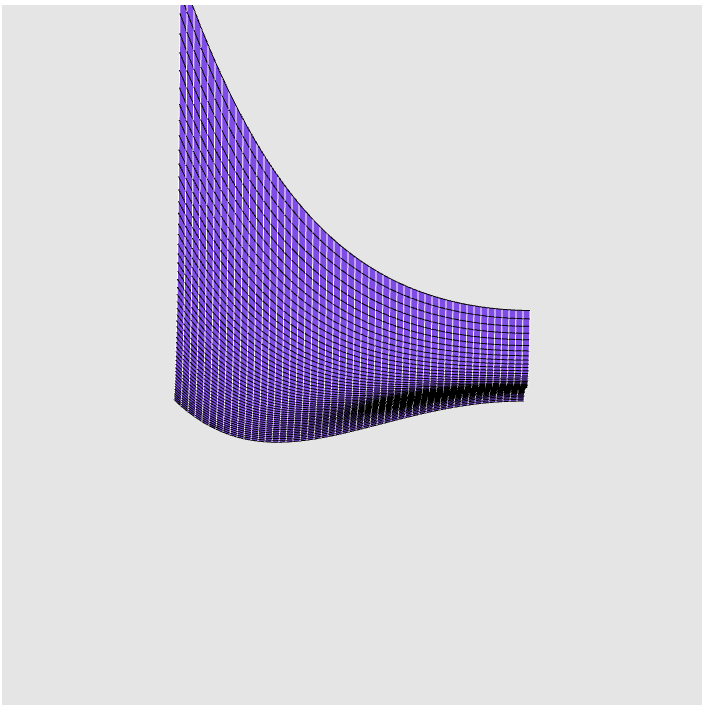
## Example Output

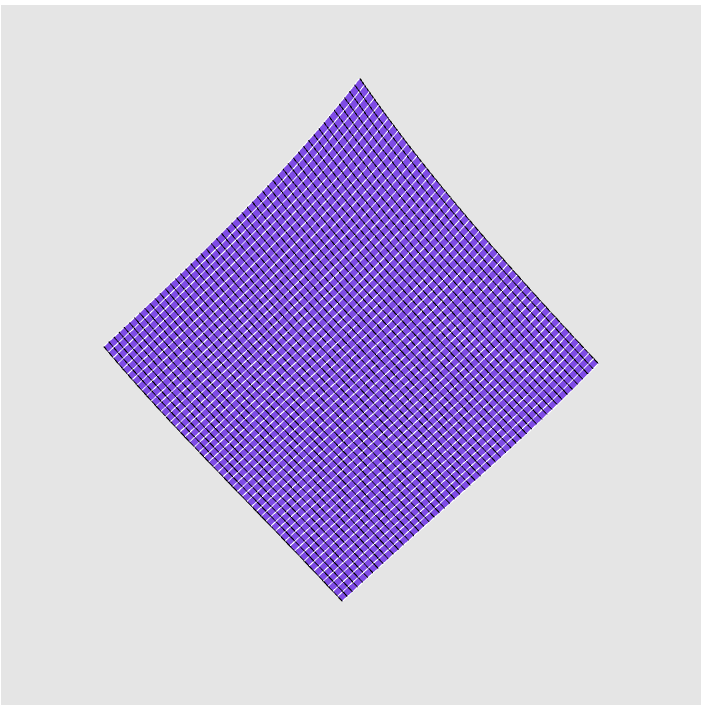












## Task 1: 3D Woodland Scene with Temple

### Code Structure

The code for the woodland scene is spread across three separate javascript files. woodland.js contains the main bulk of the code for initialising and rendering objects as well as interaction. shapes.js defines a variety of basic 3D shapes that are used in composite elsewhere: cones, cubes, and cylinders. tree.js describes the shapes needed to render a Tree object.

The code is structured in woodland.js such that global WebGL variables are defined first, followed by useful variables for calculating the appropriate views and a variety of constants used throughout the program. The grass and temple base, as simple rectangles, are defined here as well. The bulk of the WebGL initialisation code as well as generation of trees and temple follows in the init() function. Helper functions are defined subsequently at the bottom of the file: render(), which calculates model-view and projection matrices before sending them to the GPU for rendering; handleKeyDown(), which handles the keyboard-based user interaction; and generatePaths(), generateTrees(), and generateTemple(), which generate the various objects in the scene’s attributes.

### Solution Approach

In general, my solution closely follows the *desert* example in structure. The bulk of the code involves generating vertices for the one-off objects like the roof and appropriately placing the other temple objects in the scene. For the most part, this uses values calculated from the specified sizes of each temple segment, but some degree of ‘artistry’ is involved with ensuring that each portion is in a sensible location. The trees’ locations, angles, and scales are generated randomly, with some logic to ensure they do not spawn too close to the temple or other trees to avoid overlap issues. Paths are located a very slight z elevation above the grass, and similarly again for the base of the temple, to avoid ‘flickering’ from rounding errors.

To render the scene, all required objects are loaded into a single array buffer with appropriate offsets calculated. The generic objects (Cylinders, Cones, and Cubes) are loaded only once and reused across their many sources. A model-view matrix is calculated with fixed up value; up is angled such that the z-axis points up, as required. The projection matrix provides for a perspective projection.

User interaction is handled by handleKeyDown(event) which reacts to keyboard input by adjusting the viewing angle (specifically, the at and eye values are adjusted by small amounts).

### User Interface

**W, S:** Move view forwards or backwards  
**A, D:** Move view left or right in a crab-like fashion  
**Q, E:** Turn left or right in place

## Example Output

