|  |  |
| --- | --- |
| Design Document: Fractal Viewer | Jerry Guo  ICS4UI |

|  |
| --- |
| Description of the Program: |

Fractal Viewer is a ray-marching 3D graphics engine. It can render 3 different scenes:

* An infinite grid of spheres
* A Sierpinski tetrahedron
* A Mandelbulb fractal

Ray marching is a 3D rendering technique similar to ray tracing. A camera object emits points which determine the brightness of each pixel using the following process:

* The distance from each point to the closest point on the scene is estimated.
* This determines the largest “safe” distance to “march” the point.
* The point is moved along its ray by this distance.
* These steps are repeated until the ray reaches the scene or a max number of iterations.
* The brightness of a point is determined by the number of iterations required to reach a point. (The number of iterations is roughly proportional to how complex the surface of the scene is. This approximates real lighting by ensuring that more complex surfaces are shaded darker, as there are fewer possible paths for ambient light to reach it. This effect is called ambient occlusion.)

You can read more about ray marching on Mikael Christensen’s blog, which was one of the most valuable sources for me while developing this project: <http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/>

Fractal Viewer does not simulate light sources or shadows.



|  |
| --- |
| Features: |

The program includes 3 viewable scenes:

* Infinite sphere grid
* Sierpinski tetrahedron
* Mandelbulb

The fractal being rendered can be chosen in the dropdown menu.

On the right, one can set the position values for the camera, as well as the rotation values. For ease of use, one can use keyboard shortcuts to adjust the camera placement. Horizontal movement (XY plane) is mapped to WASD, with SPACE and SHIFT to move up and down (Z axis), respectively. The rotation of the camera can be adjusted using the arrow keys.

The user can also choose the movement speed of the camera. This has two modes, fixed speed and proportional speed.

* Fixed speed moves the camera by a fixed amount every frame the movement keys are held down.
* Proportional speed alters the camera’s movement speed based on its distance from the fractal. This can be set as a percentage of the distance from the fractal. This is useful for zooming in on a fractal without overshooting.



The program also allows the user to set the following graphical settings:

* Resolution
* This changes the number of rays emitted from the camera per frame. Increasing this will result in superior image quality, but will cause the program to slow down considerably.
* Field of view
* Increasing the field of view will increase the amount of visible space. However, this will also cause the outer parts of the screen to become increasingly distorted.
* Render distance
* This will stop rays after they reach a certain distance from the camera. Turning this down may make the program run smoother, but if turned too low, it will cull part of the scene. This is most noticeable in the sphere grid scene.
* Max step count
* This determines the maximum number of steps each ray will take before assuming that it hits nothing. The lighting is also based on this variable, since the brightness of each pixel is determined by the proportion of its step count and the maximum step count. Turning this value very high will slow down the program and cause the scene to become brighter. Turning it too low may cause artifacts in the image.
* Snap threshold
* Instead of requiring the distance between the ray and the fractal to be exactly 0, rays with distance under the snap threshold will also count as reaching the fractal. This is useful if the distance estimator function is not exact, since as the ray approaches the surface, the amount it marches will become less and less, and the level of relevance per iteration reaps diminishing returns. If this is set too high or too low, artifacts from overshooting or undershooting will appear.
* Safety factor
* Artifacts may occur if the ray overshoots the surface of the fractal. To prevent this from happening, the ray march distance is multiplied by a safety factor. A factor of 1 leaves the ray march distance unadjusted. The lower the factor, the more accurately the scene is rendered. This will also cause the scene to become darker, since it will take each ray more marches to reach the fractal.

There are also individual settings for the fractals:

* Iterations:
* Used for the sierpinski tetrahedron and the mandelbulb. This is how many times the fractal is iterated.
* Spacing (Sphere Grid):
* This changes the size of the spheres and the spacing between them.
* True Tetrahedron (Sierpinski Tetrahedron);
* If this is checked, the program will render tetrahedra. If not checked, it will simply draw small spheres at the required locations. At high iteration values, the difference is very small, and drawing spheres is faster. At low iteration values, true tetrahedra is necessary or else you will get a lot of empty space.
* Power (Mandelbulb):
* This determines the power variable in the mandelbulb formula. This is analogous to the ^2 term in the Mandelbrot set’s Z2 + C formula. Changing this will change the shape of the mandelbulb.

|  |
| --- |
| Coding Structure: |

**Fractal Class**

|  |  |
| --- | --- |
| Fields | |
| PVector origin | The origin of the fractal. This is where the position of the fractal is calculated from. |
| String fracType | A string that indicates the fractal type. |
| int iter | The number of times the fractal is iterated. |

|  |  |
| --- | --- |
| Methods | |
| float dE( PVector p) | Estimates the distance from a point to the fractal. In the base fractal class, this prints and error message. |

**Sphere Class (extends from Fractal)**

|  |  |
| --- | --- |
| Fields | |
| float sep | The separation between spheres. This changes the modulus function used to duplicate spheres along the grid. |

|  |  |
| --- | --- |
| Methods | |
| float dE( PVector p) | This overrides the base dE function. This estimates the distance from p to the fractal. |

**Sierpinski Class (extends from Fractal)**

|  |  |
| --- | --- |
| Fields | |
| boolean tet | If tet is true, actual tetrahedra are rendered. If this is false, spheres are drawn instead, arranged in a sierpinski tetrahedron shape. At high iterations (iter > 10), there is little visible difference, but spheres require less calculation and should run faster. |

|  |  |
| --- | --- |
| Methods | |
| float dE( PVector p) | This overrides the base dE function. This estimates the distance from p to the fractal. |

**Mandelbulb Class (extends from Fractal)**

|  |  |
| --- | --- |
| Fields | |
| int power | This affects the shape of the fractal. The classic mandelbulb uses a power value of 8. |

|  |  |
| --- | --- |
| Methods | |
| float dE( PVector p) | This overrides the base dE function. This estimates the distance from p to the fractal. |

**Camera Class**

|  |  |
| --- | --- |
| Fields | |
| PVector pos | The camera’s position. |
| PVector yaw | A unit vector representing the rotation of the camera on the XY axis. This is by default facing the Y direction. |
| float pitch | A float value representing the vertical rotation of the camera. This is constrained between π/2 (straight up) and -π/2 (straight down). |
| float speed | The fixed movement speed value. Position is added to |

|  |  |
| --- | --- |
| Methods | |
| float castRay(Fractal f, PVector p, PVector d) | Casts a ray. This ray is marched until it either hits the fractal f, exits the render distance, or hits max iterations. This then returns a float value between 0 and 1 based on the number of iterations it took to reach the fractal. The more iterations it took, the closer the returned value is to 0. This is used to determine the brightness of each pixel. |
| void update() | Gets called every frame. Calls doMoveInputs() and setGUIText(). Then, this generates rays corresponding to each pixel on the screen. This then uses castRay(); to get the brightness values for the rays, and then draws them to the screen. |
| void convertRotation(PVector d) | This takes a direction vector d and rotates it using yaw and pitch. This is used to get the rays generated by update() to actually face the camera direction. |
| void doMoveInputs() | This looks at the keys currently being pressed, and converts that into a movement vector based on yaw and pitch. For example, if the camera is rotated 90 degrees to the right, then all horizontal movements will also be rotated by that much. This allows for more intuitive controls. |
| void setGUIText() | This is just called every frame to set the values in the position fields to display the camera’s current position if the user uses keyboard controls to move. |

**Main Program**

|  |  |
| --- | --- |
| Fields | |
| int resolution | The resolution of the scene. If this is set to a nonfactor of the window size (600), this will cause the frame drawn to be smaller than the window size. |
| int renderDist | Render distance. Objects outside this distance from the camera are not drawn. |
| int iterations | Max number of ray steps. |
| float precision | Snap distance for a ray to the fractal. |
| float FOV | Field of view. The lower this is, the smaller the field of view. |
| float safety | Safety factor. This should be below 1, and the lower this is, the less distance the ray is marched per step. |
| int[] rotKeys | Stores the states of the arrow keys. This is more reliable than using the keyPressed function, which behaves strangely if the key is held down. |
| int[] moveKeys | Stores the states of the WASD, SPACE and SHIFT keys. This is more reliable than using the keyPressed function, which behaves strangely if the key is held down. |
| boolean takePicture | If this is true, the draw function will use the P versions of the graphics variables, save a picture of the drawing, then revert to the original settings. |

|  |  |
| --- | --- |
| Methods | |
| void setup() | Initializes GUI, filling it with default values.  Initializes sketch window.  Adjusts settings such as smooth() and noStroke(). |
| void draw() | If takePicture is true, this uses the P versions of all of the graphic settings variables. Then, it draws a new frame and saves it as an image, before resuming normally.  Otherwise, this just calls cam.update(). |
| void keyPressed() | This changes the values in rotKeys and moveKeys to represent the keys currently pressed. |
| void keyReleased() | This sets the values in rotKeys and moveKeys to 0 when the respective keys are released. |