WARWICK

Games Engineering

Rasterizer

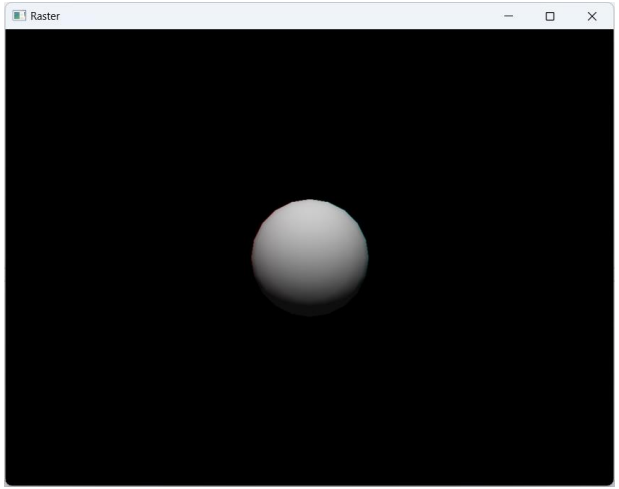
Prof. Kurt Debattista

Rasterizer

- Developed for WM9M4 Assignment
- Based on code developed for WM9M2
- Simple
  - Readable
  - Easily modifiable

Rasterizer

- Straightforward
- Only renders
  - Rectangles, cubes, sphere
- Diffuse lighting
  - Phong shaded
- Single directional light source
- Straightforward camera
- Built on Games Engineering Base



Support structs/classes

- colour (in colour.h)
  - Manipulate RGB
- vec4 (in vec4.h)
  - 4D vector (x, y, z, w)
- Vertex (in mesh.h)
  - Stores vertex information
    - Position, normal, colour

Support structs/classes

- matrix (in matrix.h)
  - 4x4 matrix
  - Transformations
- RandomNumberGenerator (in RNG.h)
  - Singleton for random numbers
- light (in light.h)
  - Light components
    - Direction
    - diffuse and ambient contributions

Support structs/classes

- Mesh (in mesh.h)
  - Colour, vertices and triangle indices of mesh
  - Mesh generation functionality
    - Rectangle, Cube, Sphere
- Z-buffer
  - Depth buffer

triangle

- In triangle.h
- Triangle for rasterization
  - Interpolates attributes
  - Rasterizes
  - Lighting
  - Depth

```
void draw(Renderer& renderer, Light& L, float ka, float kd) {
        vec2D minV, maxV;
        // Get the screen-space bounds of the triangle
        getBoundsWindow(renderer.canvas, minV, maxV);
        // Skip very small triangles
        if (area < 1.f) return;

        // Iterate over the bounding box and check each pixel
        for (int y = (int)(minV.y); y < (int)ceil(maxV.y); y++) {
            for (int x = (int)(minV.x); x < (int)ceil(maxV.x); x++) {
                float alpha, beta, gamma;

                // Check if the pixel lies inside the triangle
                if (getCoordinates(vec2D((float)x, (float)y), alpha, beta, gamma)) {
                    // Interpolate color, depth, and normals
                    colour c = interpolate(beta, gamma, alpha, v[0].rgb, v[1].rgb, v[2].rgb);
                    c.clampColour();
                    float depth = interpolate(beta, gamma, alpha, v[0].p[2], v[1].p[2], v[2].p[2]);
                    vec4 normal = interpolate(beta, gamma, alpha, v[0].normal, v[1].normal, v[2].normal);
                    normal.normalise();

                    // Perform Z-buffer test and apply shading
                    if (renderer.zbuffer(x, y) > depth && depth > 0.01f) {
                        // typical shader begin
                        L.omega_i.normalise();
                        float dot = max(vec4::dot(L.omega_i, normal), 0.0f);
                        colour a = (c * kd) * (L.L * dot + (L.ambient * kd));
                        // typical shader end
                        unsigned char r, g, b;
                        a.toRGB(r, g, b);
                        renderer.canvas.draw(x, y, r, g, b);
                        renderer.zbuffer(x, y) = depth;
                    }
                }
            }
        }
    }
```

Renderer

- renderer (renderer.h)
  - Stores rendering components
  - Z-buffer
  - Screen buffer
  - Perspective matrix

```
void render(Renderer& renderer, Mesh* mesh, matrix& camera, Light& L) {
    // Combine perspective, camera, and world transformations for the mesh
    matrix p = renderer.perspective * camera * mesh->world;

    // Iterate through all triangles in the mesh
    for (triIndices& ind : mesh->triangles) {
        Vertex t[3]; // Temporary array to store transformed triangle vertices

        // Transform each vertex of the triangle
        for (unsigned int i = 0; i < 3; i++) {
            t[i].p = p * mesh->vertices[ind.v[i]].p; // Apply transformations
            t[i].p.divideW(); // Perspective division to normalize coordinates

            // Transform normals into world space for accurate lighting
            // no need for perspective correction as no shearing or non-uniform scaling
            t[i].normal = mesh->world * mesh->vertices[ind.v[i]].normal;
            t[i].normal.normalise();

            // Map normalized device coordinates to screen space
            t[i].p[0] = (t[i].p[0] + 1.f) * 0.5f * static_cast<float>(renderer.canvas.getWidth());
            t[i].p[1] = (t[i].p[1] + 1.f) * 0.5f * static_cast<float>(renderer.canvas.getHeight());
            t[i].p[1] = renderer.canvas.getHeight() - t[i].p[1]; // Invert y-axis

            // Copy vertex colours
            t[i].rgb = mesh->vertices[ind.v[i]].rgb;
        }

        // Clip triangles with Z-values outside [-1, 1]
        if (fabs(t[0].p[2]) > 1.0f || fabs(t[1].p[2]) > 1.0f || fabs(t[2].p[2]) > 1.0f) continue;

        // Create a triangle object and render it
        triangle tri(t[0], t[1], t[2]);
        tri.draw(renderer, L, mesh->ka, mesh->kd);
    }
}
```

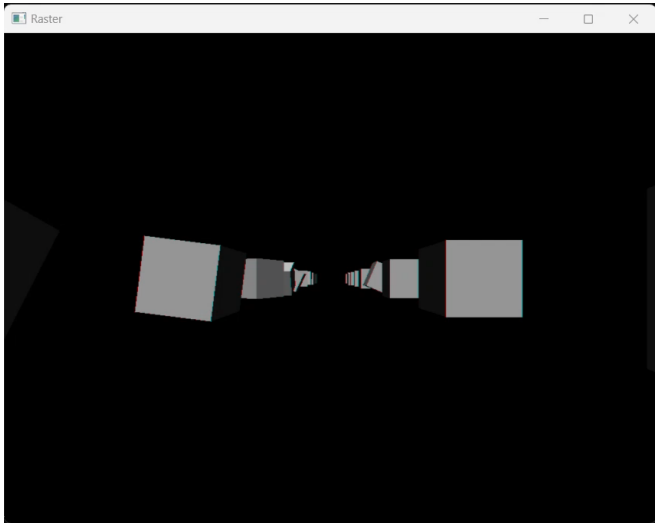
Scenes

- Scene
  - Needs to do heavy lifting
  - Create renderer, camera, lights, scene
  - Game loop
    - Handle input
    - Animate (if necessary)

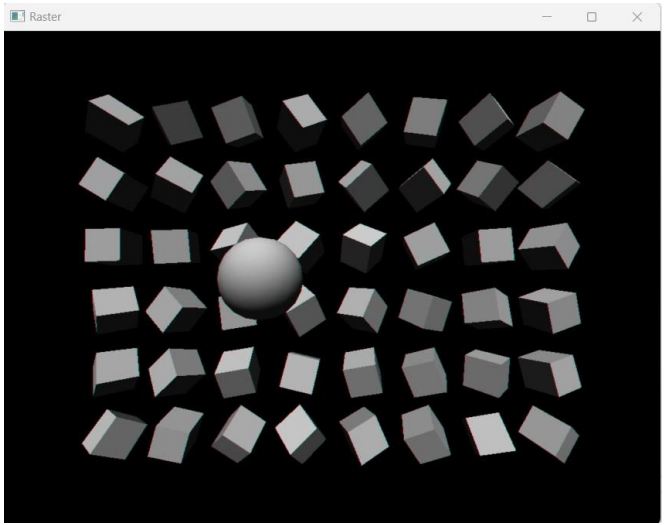
```
void sceneTest() {
    Renderer renderer;
    // create light source {direction, diffuse intensity, ambient intensity}
    Light L{ vec4(0.f, 1.f, 1.f, 0.f), colour(1.0f, 1.0f, 1.0f), colour(0.1f, 0.1f, 0.1f) };
    // camera is just a matrix
    matrix camera = matrix::makeIdentity(); // Initialize the camera with identity matrix

    bool running = true; // Main loop control variable

    std::vector<Mesh*> scene; // Vector to store scene objects
    Mesh mesh = Mesh::makeSphere(1.0f, 10, 20); // Create a sphere
    scene.push_back(&mesh); // Add to scene

    float x = 0.0f, y = 0.0f, z = -4.0f; // Initial translation parameters
    mesh.world = matrix::makeTranslation(x, y, z); // world transformation of mesh

    // Main rendering loop
    while (running) {
        renderer.canvas.checkInput(); // Handle user input
        renderer.clear(); // Clear the canvas for the next frame

        // Apply transformations to the meshes (if needed eg user moves)
        mesh.world = matrix::makeTranslation(x, y, z);

        // Handle user inputs for transformations
        if (renderer.canvas.keyPressed(VK_ESCAPE)) break;
        if (renderer.canvas.keyPressed('A')) x += -0.1f;
        if (renderer.canvas.keyPressed('D')) x += 0.1f;

        // Render each object in the scene
        for (auto& m : scene)
            render(renderer, m, camera, L);

        renderer.present(); // Display the rendered frame
    }
}
```

Scenes

Scene 1



Scene 2



Suggestions

- Keep it simple
- Do not change resolution
- Do not change renderer parameters
- Add functionality gradually
  - Test all scenes when you do so
- Think about scene 3

WARWICK

Games Engineering

Rasterizer

Prof. Kurt Debattista