A5 Final Report

Title: Realistic Bridge Defect Simulations using Ray-tracing for Synthetic Training Data

Name: Zaid Al-Sabbag
Student ID: 20561516
User ID: zaalsabb

Final Project:

Purpose :
Simulate realistic images of structural defects in bridge structures for the goal of generating synthetic images to train AI models.

Statement :
In recent years, the field of bridge inspections and structural health monitoring has started to embrace machine learning approaches to detect and classify structural defects (cracks, spalling, corrosion) in images of bridge structures [1, 2]. To train these machine learning models, we need a large database of images of structures with and without defects, varying lighting conditions, camera angles, and clutter in the scene so that the model learns to distinguish between images with and without defects. Unfortunately, it is expensive and time-consuming to build such a large database from real images. Thus, many researchers in this field have proposed to use 3D simulations to create synthetic images to augment or even replace real images when training models [3, 4, 5]. However, it is necessary to achieve a high degree of realism when creating these simulated images, as these synthetic images should be indistinguishable from real images of defects. Thus, the goal of this project will be to use ray-tracing and other advanced rendering techniques to simulate defects on bridge structures and create realistic synthetic images.

The ray-tracing scene will have a 3D model of a concrete bridge that was modified to include a spalling-type structural defect. Several objects such as road signs and traffic cones will be included in the scene as clutter to simulate a real bridge environment. The ground plane will have a road texture and parts of it may be covered in a dirt texture.

This scene will be challenging to render because it is not trivial to render real-world conditions because there are many factors to consider. The unique properties of concrete (random bumps) makes bridge structures a challenging scene to render accurately.

During this project, I plan to learn useful skills in several rendering topics such as:

1. Photo-realistic rendering of heterogeneous materials such as concrete.
2. Efficient ray-tracing of complex scenes.
3. Simulate realistic camera lens effects for computer vision applications.

Technical Outline :

Materials such as concrete do not have a uniform surface or color. Thus, texture and bump mapping will have to be implemented to simulate the surface of real concrete, and parameters for a realistic shading model will have to be selected and tuned. Perlin noise will be used to estimate the amount of bump in the concrete surface. Textures will also be used for traffic signs and traffic cones. A lens system will also have to be implemented to mimic the effects of using a real camera for computer vision applications.

Custom primitives such as cones and cylinders will also be implemented. Cone primitives will be used to model traffic cones, while cylinders will be used to model traffic signs. Quadrilateral meshing will also be implemented, since some parts of the bridge section model (defect section) use quads instead of triangles for the mesh.

Anti-aliasing will be applied to the image to make it look more realistic. Soft shadows will also be implemented to simulate the effect of real shadows as much as possible using area lights.

Acceleration data structures such as octrees will also be implemented to reduce the amount of time needed to perform raytracing calculations on complex triangle and quad meshes. Ray grouping will also be implemented to reduce processing time.

Technical resources used: [6, 7, 8, 9, 10].

Implementation :

New commands in lua file:

1. **gr.cone**
2. **gr.cylinder**
3. **gr.texture**
4. **gr.bump**
5. **gr.perlin**
6. **gr.arealight**
7. **gr.quad**

Custom primitives (cone, cylinder): For this project, custom primitives such as cones and cylinders were implemented. These are deployed using lua commands. For the cone primitive, (x,y,z) is the position of the top of the cone. For the cylinder primitive, (x,y,z) is the position of the bottom of the cylinder. Height and radius of the cone/cylinder are specified by the user. The ray-cylinder interaction equations were based on this implementation [11].

```
cone = gr.cone('cone', {x, y, z}, height, radius)
cylinder = gr.cylinder('cylinder', {x, y, z}, height, radius)
```

Files: Primitives.cpp, Primitives.hpp

Texture mapping Textures were implemented by mapping the intersection point on the object's surface to a 2D coordinate (u,v) on an image. This is performed while checking when a ray intersects with an object. The 3D coordinate is first projected onto the object's surface. Then, this point is mapped to a point on the image to obtain a color value from the image. A texture object is first created using a PNG image provided by the user:

```
sign_texture = gr.texture("sign.png")
```

Then, the texture is attached to a **Quad**, **Cone**, or **Cylinder** node using the following lua command:

```
sign = gr.cylinder('pole', {x,  y, z}, height, radius)
sign:set_texture(sign_texture)
```

Files: Texture.cpp, Texture.hpp

Bump Mapping Bump mapping was implemented by importing a PNG image containing the depth information of the associated heightmap of the surface. The intersection normal with the surface is then modified based on the gradient of the heightmap using the equation from this implementation [12]. The user can add a Bump object to a Quad object using the following lua command:

```
spalling_bump = gr.bump("wall_depth.png")
```

Then assign the Bump object to a Quad object:

```
wall = gr.quad( 'wall', 'quad.obj')
wall:set_bump(spalling_bump)
```

Files: Bump.cpp, Bump.hpp

Lens system (depth of field) The depth of field was implemented by using a Circle of Confusion (CoC) [13] region where the values of pixels are averaged. The CoC value for each pixel is calculated using a depth buffer and three parameters: Aperture constant (A), focal length (f), and screen position (s). These parameters are specified in the gr.render command:

```
gr.render(node, filename, w, h, eye, view, up, fov, ambient,
point_lights, area_lights, {A,f,s}, super_resolution_factor)
```

Files: A5.cpp, A5.hpp

Ray-Quad Intersection A quad mesh object was implemented to compliment the triangular mesh. This makes it easier to implement features such as texture mapping and reduces the number of mesh objects to check intersections by half. The Quad object is initialized as follows:

```
wall = gr.quad( 'wall', 'quad.obj')
```

The quad.obj file looks like this:

```
o quad

v -1 -1 0
v 1 -1 0
v 1 1 0
v -1 1 0

f 1 2 3 4
```

Files: Quad.cpp, Quad.hpp

Anti-aliasing In this project, anti-aliasing was implemented using super-resolution, then averaging of the pixels of each block. The user can set the size of the block (S) of the super-resolution in the gr.render lua command:

```
gr.render(node, filename, w, h, eye, view, up, fov, ambient,
point_lights, area_lights, depthfield, S)
```

Files: A5.cpp, A5.hpp

Perlin Noise Perlin noise was implemented using a pre-generated random seed vector of size 512. The seed vector is then used to generate a heightmap of custom resolution decided by the user. The implementation and original algorithm are described here [14] and here [15]. The perlin heightmap is then treated as a bump map and is used to modify surface normals based on calculated gradient. These are then used to perturb the intersection normals. The user first creates a perlin object by specifying the noise amplitude (a), octave (o), and resolution (w,h).

```
road_perlin = gr.perlin(a,o,w,h)
```

Files: Perlin.cpp, Perlin.hpp

Soft shadows (area light) Soft shadows are implemented by sampling a light source multiple times. and the light source's position is randomly moved inside a radius (R) specified by the user. The user also specifies the number of samples (N). The brightness is then averaged over N samples.

```
arealight = gr.arealight({x,y,z}, {r,g,b}, {1, 0, 0}, R, N)
```

The user then adds the list of area light sources to the gr.render lua command:

```
gr.render(node, filename, w, h, eye, view, up, fov, ambient,
point_lights, area_lights, {A,f,s}, super_resolution_factor)
```

Files: Light.cpp, Light.hpp, A5.cpp, A5.hpp

Acceleration data structure for triangle/quad meshes (octree) Unfortunately this objective was not implemented due to limited time.

Final scene (bridge structure with spalling, car, cones, and traffic sign) The final rendered scene can be found in **A5/screenshot.png**. The car was not included in this scene because none of the complex models I found online were compatible unfortunately because they use both quads and triangles in the same mesh, which is not supported by my implementation.

Graduate Report :

This report fulfills the requirement of CS 688 graduate students. It covers the literature sources of three topics used in this project: Bump Mapping, Depth of Field, and Perlin Noise.

Bump Mapping The concept of bump mapping was first introduced in 1978 by [16] to create nonuniform rough surface by perturbing surface normals to create the illusion of a nonuniform surface. This can be achieved by first calculating the gradient of the bump map (D):

$$F_u = \frac{D[u+1,v] - D[u-1,v]}{2}, F_v = \frac{D[u,v+1] - D[u,v-1]}{2} \tag{1}$$

The perturbed normal (N') is calculated using the original normal (N), local orthonormal vectors of the surface (P_u, P_v) and bump map gradient (F_u, F_v):

$$N' = N + \frac{F_u(n \times P_v) - F_v(n \times P_u)}{|N|} \tag{2}$$

Perlin Noise Perlin noise is a technique used in computer graphics to generate procedural noise. The tecnique computes a pseudo-random gradient at every point along the surface, which is then used to compute a heightmap. In the improved Perlin noise implementation [15] the fade function $3t^2 - 2t^3$ is replaced by $6t^5 - 15t^4 + 10t^3$. This gurantees that the heightmap has a first and second order derivatives of 0 at t=0 and t=1. Thus, guaranteeing a continuous map. Another factor to consider is the number of octaves used to generate the map. Higher octaves create a more detailed map, while lower octaves create smoother maps.

Depth of Field Depth of field is the effect that cameras create when objects that are in-focus appear sharper and less blurred than objects that are out-of-focus. Thus, Depth of field is defined as the distance between the camera and the closest object that appears sharp. To simulate this real effect in a virtual scene, a simple implementation is to use a Circle of Confusion (CoC) to artificially blur objects that should be out-of-focus [13]. Several parameters need to be defined to calculate the CoC radius, such as depth between the camera and object in pixel (d), focal length (f), distance between aperture and screen (s), and aperture size (A). In this implementation, the CoC radius (R_{CoC}) is defined as the following:

$$R_{CoC} = |A(s(1/f - 1/d) - 1)| \tag{3}$$

The CoC radius will vary per pixel depending on the distance between camera and objects in each pixel. A filter is then used to iterate over every pixel (p_i), calculate R_{CoCi}, then calculate the average brightness value of all pixels inside R_{CoCi} and assign that value to p_i. Thus, pixels with high d values will be more blurred than pixels with lower d, which achieves the result of simulating the real depth of field produced by cameras.

Bibliography :

[1] Billie F Spencer Jr, Vedhus Hoskere, and Yasutaka Narazaki. Advances in computer vision-based civil infrastructure inspection and monitoring. *Engineering*, 5(2):199–222, 2019.

[2] Vedhus A Hoskere. *Developing autonomy in structural inspections through computer vision and graphics.* PhD thesis, University of Illinois at Urbana-Champaign, 2020.

[3] Vedhus Hoskere, Yasutaka Narazaki, and BF Spencer. Learning to detect important visual changes for structural inspections using physics based graphics models. In *9th International Conference on Structural Health Monitoring of Intelligent Infrastructure: Transferring Research into Practice, SHMII 2019*, pages 1484–1490. International Society for Structural Health Monitoring of Intelligent . . . , 2019.

[4] Ricardo Silva Peres, Magno Guedes, Fábio Miranda, and Jose Barata. Simulation-based data augmentation for the quality inspection of structural adhesive with deep learning. *IEEE Access*, 9:76532–76541, 2021.

[5] Ricardo Silva Peres, Miguel Azevedo, Sara Oleiro Araújo, Magno Guedes, Fábio Miranda, and José Barata. Generative adversarial networks for data augmentation in structural adhesive inspection. *Applied Sciences*, 11(7):3086, 2021.

[6] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[7] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[8] Andrew S Glassner. *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[9] Peter Shirley and R Keith Morley. *Realistic ray tracing*. AK Peters, Ltd., 2008.

[10] Steve Marschner and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2018.

[11] Julien Guertault. Intersection of a ray and a cone, Jan 2017.

[12] Anders Hast. Bump mapping. 01 2008.

[13] Tin-Tin Yu. Depth of field implementation with opengl. *Journal of Computing Sciences in Colleges*, 20(1):136–146, 2004.

[14] Raouf Touti.

[15] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[16] James F Blinn. Simulation of wrinkled surfaces. *ACM SIGGRAPH computer graphics*, 12(3):286–292, 1978.

Objectives:

Full UserID: zaalsabb **Student ID: 20561516**

___ 1: Custom primitives (cone, cylinder).

___ 2: Texture mapping.

___ 3: Bump mapping.

___ 4: Lens system (depth of field).

___ 5: ray-quad intersection.

___ 6: Anti-aliasing.

___ 7: Perlin noise.

___ 8: Soft shadows (area light).

___ 9: Acceleration data structure for triangle/quad meshes (octree).

___ 10: Final scene (bridge structure with spalling, car, cones, and traffic sign).

A4 extra objective: reflections and refractions.