Development of a Graphical Interface for an Endoscopy Simulator

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Engineering
IN
Computer Science and Engineering

BY

Tarun Bansal

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2015

Declaration of Originality

I, **Tarun Bansal**, with SR No. **04-04-00-10-41-13-1-10276** hereby declare that the material presented in the thesis titled

Development of a Graphical Interface for an Endoscopy Simulator

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2014-15**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

My Parents

Acknowledgements

I am thankful to everyone who was there to help and support me during the course of my project and the writing of this thesis. First I would like to thank my project guide Dr. Vijay Natarajan, who offered me this project despite knowing the fact that I did not have significant background in computer graphics at that time. This thesis would not have been possible without his able guidance and patience. I would also like to thank Prashanth Uduppa and Sudarshan Rao from BigSolv Labs for their patience and support. I thank Shantanu and Gokul for helping me in early days of my project work. I also thank the reader Dr. Uday Kumar Reddy B., for his thorough and careful review of this work and for valuable suggestions. I would like to thank my lab mates for their helpful suggestions during lab talks. I especially thank Talha for his constant support and advices. I also thank Pushparaj for fixing various issues that kept on occuring with my system during my project work. I would also like to thank all my classmates and colleagues for their unconditional support throughout my course.

I thank my parents who always supported me in every crucial decision of life. I thank my family and friends for all their support.

Abstract

Graphical interface is a very important component of an endoscopy simulator that contributes in its success. We have developed a graphical interface that can be integrated with mechanical system of endoscopy simulator. As a part of this work, we have developed a technique for realistic rendering of the human tissue, technique for texture mapping for stomach and GI tract. We have also worked out features like collision detection, tube rendering, mucous placement and presence of Z- line. This realistic rendering helps doctors to get the more real feeling during their training in simulated environment. We have improved rendering using normal mapping, gloss mapping and lapped texturing on programmable rendering pipeline.

Contents

List of Figures

Chapter 1

Introduction

1.1 Endoscopy Simulator

Demand of endoscopy simulator has been increasing among doctors because of its safe and effective training on various procedures. It is cheaper, time saving and can be used in repeated practices. It has power to evaluate performance of a trainee doctor. In this simulator there are two kinds of feedbacks available for a trainee, one is called force feedback generated by a mechanical device known as haptic device and another one is called visual feedback generated by underlying graphics system. Both mechanical and graphical systems are integrated as one complete system.

1.2 Graphical Interface for an Endoscopy Simulator

Success of endoscopy simulator is heavily dependent on visual feedbacks. Visual feedbacks are nothing but what a trainee sees on the screen during a training session. Ideally the graphics system should mimic the actual human body by giving the best possible visual feedbacks. Visual feedback requires realistic rendering of wet tissue, red out in case of collision, tube appearance when camera is turning back, presence of Z-line and mucous. Different kind of realism may be needed due to variation in tissue structure, for example appearances of upper GI tract and stomach are not same in the actual endoscopy procedure.

1.3 Role of Graphics Hardware in Graphical Interface development

In nineties the rendering pipeline used to be fixed without giving any control to developer over any of its stage. At that time it was almost impossible to implement any good quality

rendering schemes because lighting and shading calculations were completely based on vertices of model that is going to be rendered. Due to heavy demand in computer gaming industry, rendering hardware has undergone many major changes in last more than ten years. Nowadays, almost every graphics hardware vendor is building hardware with programmable pipeline. Programmable pipeline has made possible to do lighting and shading calculations for each fragment and thus rendering quality has improved drastically. It has also fuelled the research in photo-realistic rendering for surgical simulators [9]. In this work we are using programmable pipeline for graphics interface development.

Chapter 2

Related Work

2.1 Photo realistic rendering

Photo realistic rendering of tissue is still an active area of research in computer graphics domain. Stoyanov et al. [10] present a review of some techniques required for building patient specific models using geometric information about the scene.

Elhelw et al. [5] talks about the use of reflection map and refraction map for enhancing realism of wet surfaces. Reflection map is the way of storing perturbed normal for doing specular lighting calculation. Refraction map is a model of achieving refraction effects because of wet surface formation over the tissue. Visser et al. [11] addresses the similar problem of visualization of colonic lumen using some techniques like normal mapping and gloss mapping. Detailed description of normal and gloss mapping is given in sections of coming chapters.

2.2 Texture mapping for complex shaped objects

Praun et al. [8] talks about a method for texture mapping for an arbitrary surface mesh using an example 2D texture. It repeatedly pastes texture patches on the surface patches until the whole mesh is completely covered. A surface patch is a small area on the surface of the 3D model suitable to map a single texture patch.

We employ this method with improvements to adapt it to the GI tract and stomach model. Detailed description of our technique is given in sections of coming chapters.

Chapter 3

Background

3.1 Model

It is the representation of entire surface in terms of connected polygons. It has essentially a list of all vertices and a list of all face polygons. Usually polygons are triangles and that model is called triangular mesh. Triangular mesh for a sphere is shown in Figure 3.1.

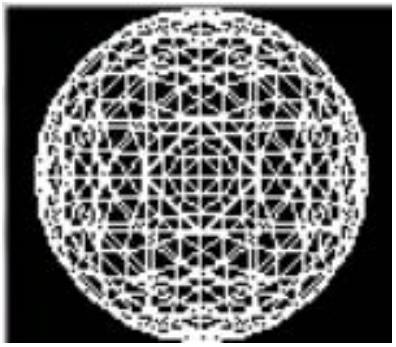
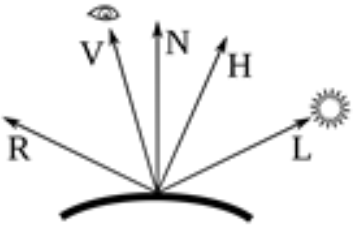


Figure 3.1: Triangular mesh for a sphere.

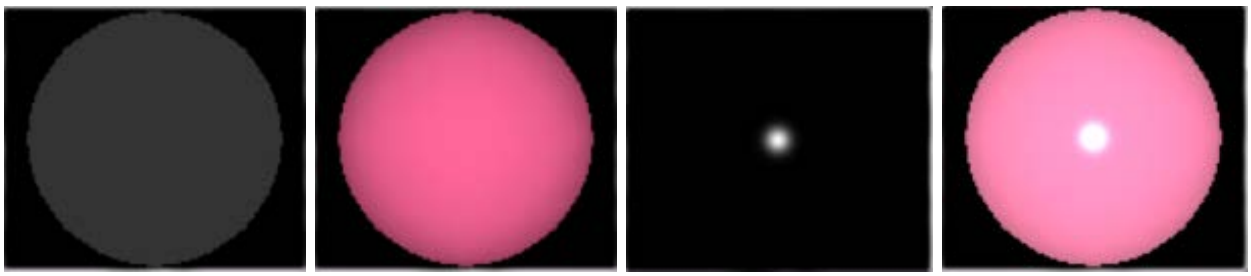
3.2 Lighting and shading

Lighting is necessary for giving 3D effect. There are different schemes available for lighting calculation. One such scheme was proposed by Blinn [4]. In this scheme, four vectors are involved in lighting calculation at a given surface point. These vectors are shown in Figure 3.2(a), incoming light direction L, surface normal N, halfway vector H and eye direction V. H is halfway vector between V and L. Final colour involves three separate components namely ambient, diffuse and specular. Ambient colour is the result of global illumination and for simplicity it is kept constant for the entire model. Diffuse and specular colours depend on

object and light properties. Additionally diffuse intensity is proportional to $L.N$, where $L.N$ is dot product of L and N. Specular intensity is proportional to $(H.N)^{\eta}$ where $(H.N)$ is dot product of H and N and η is the shininess coefficient of the material. A sphere with different lighting effect is shown in Figure 3.2.



(a)

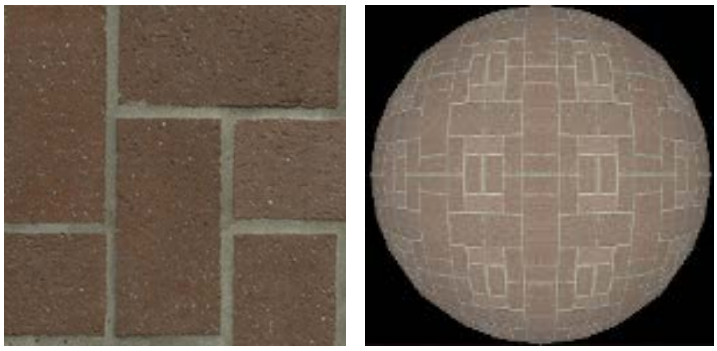


(b) (c) (d) (e)

Figure 3.2: Lighting and shading demonstrated on a sphere. (b) A sphere with only ambient lighting. (c) Same sphere with only diffuse lighting. (d) With only specular lighting. (e) The combined effect of all three lighting. (a) The vectors involved in lighting calculation. L is light vector, V is viewing vector, N is normal to the surface, R is reflection vector, while H is halfway vector between V and L.

3.3 Texturing

This technique is used for giving more details to the surface. Texture is an image that can be pasted on every face of model during rendering. Texture mapping is a technique for mapping different parts of texture on different faces. Each vertex of a face is given 2D texture coordinates (u, v) which is some point inside texture image. These texture coordinates specify the part of the texture that will get pasted on that particular face. A texture image and a sphere with texture mapping is given in Figure 3.3.



(a) Given texture (b) Textured sphere

Figure 3.3: An example of texture mapping.

3.4 Programmable Pipeline

Nowadays graphics hardware comes with programmable rendering pipeline. This pipeline has many stages and some stages are programmable. Shaders are used to program these stages. Shaders are programs written in GLSL [1], which is a high level programming language. It provides direct control of rendering pipeline to a developer. Two important shaders are vertex shader and fragment shader as shown in the Figure 3.4. Vertex shader runs for each vertex and fragment shader runs for each fragment of all the faces. Shaders are bound to pipeline stages by making OpenGL [3] function calls in the application program.

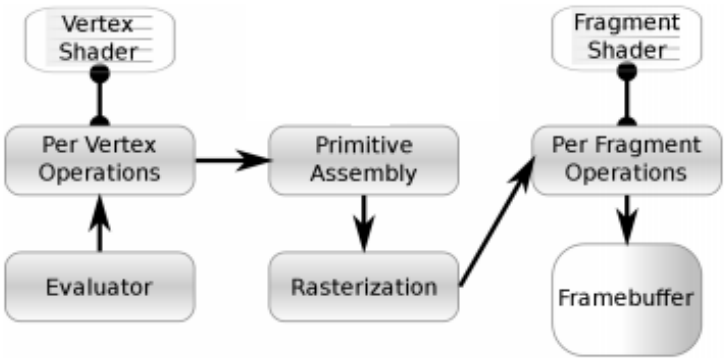


Figure 3.4: Stages of programmable graphics pipeline.

Whatever we see on the screen, is actually the projection of the object on the flat plane. For achieving this projection, object is transformed in different coordinate systems by multiplying it with different matrices. Every vertex is given in model coordinate system in the model (mesh). In the first stage parameters of vertices are evaluated by running OpenGL commands and theses are send in the next stage for vertex processing. In vertex processing stage vertex

parameters are transformed in camera space from model space. The matrix involved in this step is called ModelView matrix. Object is also projected on the flat plane using Projection matrix. The corresponding shader for this stage is called vertex shader. The next stage is primitive assembly where projected vertices are joined to form. In the next stage, primitives are rasterized into pixel fragments and parameters for each pixel fragment are calculated by interpolation of parameters of vertices of primitive. In the final stage, lighting calculation is performed for each fragment. The shader associated with this stage is called fragment shader.

Chapter 4

Proposed Techniques

4.1 Overview

Rendering of tissue for graphical interface should contain some specifics of GI tract and stomach like mucous presence, glossy look, Z-line presence, etc. We use lapped textures [8] that provides good solution of stomach and GI tract texturing. We use techniques like normal mapping and gloss mapping that are used for calculation of specular highlights. Ambient and diffuse colours are calculated using standard calculations.

Application program reads the entire model stored in OFF [2] file. After reading entire model, various vertex attributes like normal, texture coordinate, tangent, etc. are calculated and transferred in the graphics hardware's memory. These attributes are used in vertex and fragment shader for calculating final colour using texture, normal and gloss mapping. After transferring all the vertex attributes, graphics pipeline is called which is used for collision detection as well as generation of coloured frame for model. If there is a collision, entire screen is rendered as red otherwise model is displayed together with endoscopy tube. All techniques like texture mapping, normal mapping, gloss mapping, tube rendering, collision detection, etc. are explained in coming subsections and detailed process of rendering is explained in Section 4.9.

4.2 Lapped texture mapping

This technique is used for generating texture coordinates for vertices. We cannot use simple texturing here because of complicated stomach and GI tract model. Figure 4.1 shows the effect of simple mapping in the GI tract.

We solve this problem by employing lapped textures [8] with improvements. In this technique we divide entire surface into many patches such that each patch is almost flat and circular.

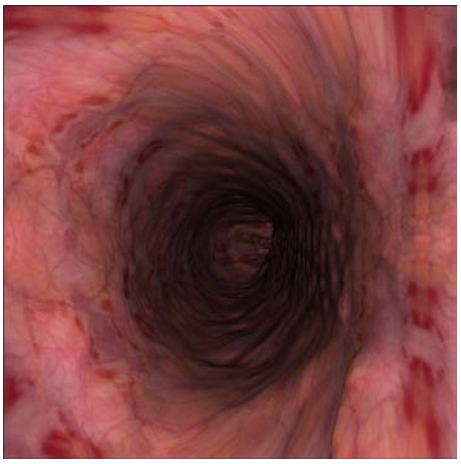


Figure 4.1: Plane texturing of GI tract.

As each patch is almost flat, it can be textured without stretching of texture.

4.2.1 Dual graph generation

Here first we create dual graph G of mesh M. G has same number of nodes as there are triangles in M. Each node in G can have at-most three edges which represent three adjacent triangle sharing edges of corresponding triangle in M. Figure 4.2 shows the example mesh and its dual graph. Following algorithm is used for generate dual graph (G) -

Algorithm 1 Create Dual Graph

Input: M: Triangular Mesh

Output: G: Dual Graph

1: Create empty list E.
2: For each triangle t, insert $< u, v, t >$ into E for every edge (u, v).
3: Sort all the triplets in lexicographic order.
4: Make an empty graph G in which number of nodes is equal to number of triangles in M.
5: For every repeated triplet with two different t_i and t_j, make an edge between n_i and n_j in G.
6: **return** G

4.2.2 Patch generation

After calculating G we start growing patch around randomly selected seed triangle. A triangle is included if it increases the area of convex hull of projection of patch in the plane of seed triangle. Following algorithm is used for checking inclusion criteria-

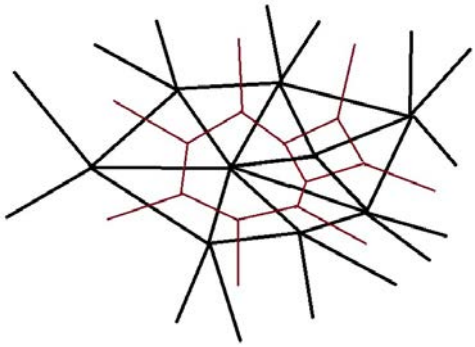


Figure 4.2: A triangle mesh (black lines) and its dual graph (red lines).

Algorithm 2 inclusionTest

Input: x, s, M: Testing node and seed node

Output: true or false

Get t_x from M which is corresponding to x in G.
Get t_s from M which is corresponding to s in G.
Find angle θ between normals of t_x and t_s.
If angle $\theta <$ (90 - tolerance)
    return true
else
    return false

We grow patch around the seed triangle until distortion reaches some predefined tolerance. Triangles are added in patch in Breath First Search order starting from seed in G. A triangle that satisfies inclusion criteria is considered for inclusion in current patch. If a triangle does not satisfy inclusion criteria, we start counting the distortion and with every level of triangles traversed, we increase it by 1. We do this until distortion reaches predefined tolerance value, after that we stop growing patch. While growing patch we assign texture coordinates to each newly added triangle. Figure 4.3 shows patches where black triangle shows seed for that patch. Following algorithm is used for generating patches-

Algorithm 3 Create Patches

Input: M: Triangular Mesh

Output: Modified M with texture coordinates

Generate G from M using Algorithm 1.
Make a random permutation S of nodes in G.
For every uncovered node n in S do following
    select n as seed s.
    call growPatch(s,G).
return M

Algorithm 4 growPatch

Input: s: Seed

Input: G: Graph

Create a patch as a list P containing s.
Get t_s from M which is corresponding to s in G.
Set distortion as 0.
Set circular flag true.
Create empty queue Q. Insert s in Q. Make s covered.
    Deque node n from Q.
    Get t_n from M which is corresponding to n in G.
    Texture t_n by projecting it into plane of t_s.
    If distortion $<$ tolerance,
        For every uncovered neighbour x of n
            if inclusionTest(x,s) is true
                push x in Q and make x covered.
                if no more circular then set distortion as distance between last circular triangle and x.
            else
                if circular flag true then make it false and remember x as last circular triangle.

4.2.3 Straight line artifacts

This kind of texturing causes straight line artifacts along the boundaries of patches. For overcoming this artifacts, we use blending. We assign two set of texture coordinates to every border

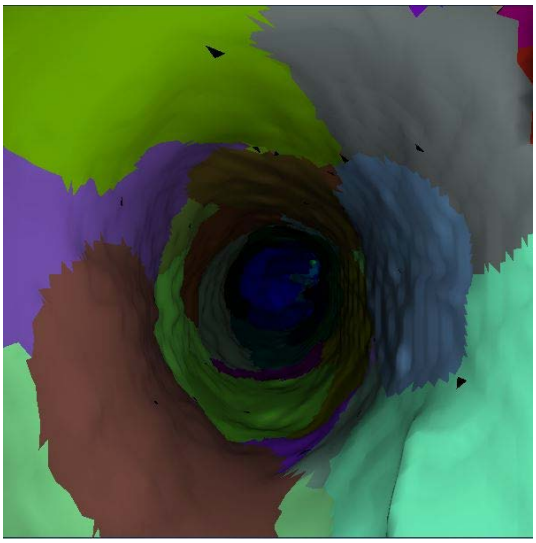
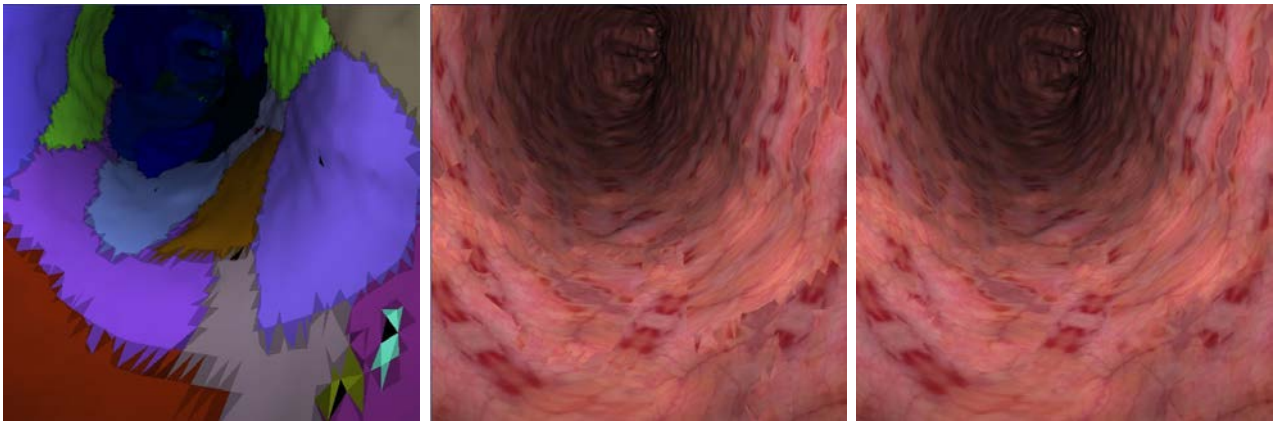


Figure 4.3: Patches for GI tract (black triangle is a seed triangle).

triangle. One comes from parent patch and another from adjacent patch. Final colour will be blended colour of two colours coming from two texture coordinates. Figure 4.4 shows straight line artifacts and overcoming it by blending.



(a) (b) (c)

Figure 4.4: Straight line artifacts generated by lapped textures. (a) Patches used in texturing. (b) Actual texturing without blending. (c) Actual texturing with blending.

4.2.4 Blending factor

As all calculations happen in fragment shader, so blending happen for every fragment. We need to have a blending factor (α) for calculating share of every colour in final colour. α for every fragment of a triangle is calculated using interpolation of α of all three vertices during

rasterization stage. We need to specify α carefully for each vertex. Specifying α as 0.5 for all three vertices of two boundary triangles will eliminate existing artifact occurring on the shared edge but create four more dull artifacts along remaining four edges. Specifying α as 0.5 for two shared vertices and 1.0 for non shared vertex of two boundary triangles will eliminate existing artifact occurring on the shared edge and also minimize other four dull artifacts occuring along remaining four edges. Figure 4.5 shows effect of different α.

(a) (b) (c) (d)

Figure 4.5: Effect of different α. (a) α specified at every vertex of traingle. (b) Artifact without blending. (c) Effect of $\alpha = 0.5$ at every vertex. (d) Effect of $\alpha = 0.5$ for shared vertices and $\alpha = 1.0$ for non shared vertex.

4.2.5 Duplication of vertices

Here we need to duplicate the boundary veritces in order to get the correct texturing. Figure 4.6 shows the result of not duplicating them.

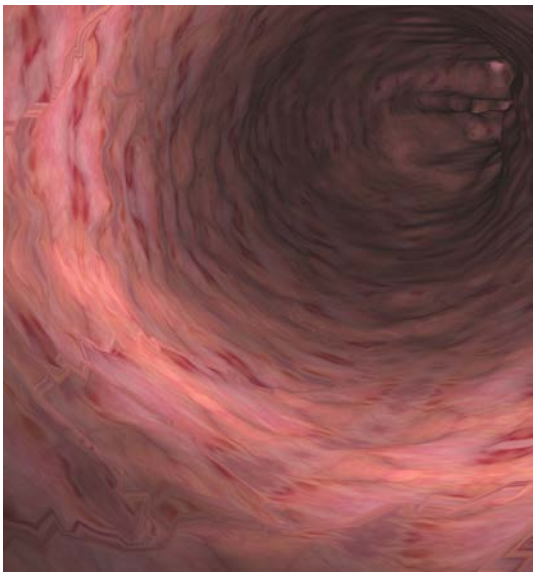


Figure 4.6: Texturing without duplicating vertices.

4.3 Normal mapping

This technique is used for generating perturbed normals. As the tissue in stomach and GI tract is highly reflective, this kind of effect cannot be captured if lighting calculations are done using regular normals. We solve this problem by using normal mapping [6].

In programmable pipeline, lighting calculations are done for all the fragments in fragment shader. After rasterization, each fragment has its own normal and texture coordinates. Normal is used for lighting calculation and texture coordinate is used for finding corresponding texel in texture. We need perturbed normal for each texel in texture image.

4.3.1 Generation of perturbed normals

Normals are generated using following procedure-

1. Take a grid of equal dimensions as the texture image.
2. Calculate height for each grid point by Perlin noise [7].
3. Now treat this grid (with height at all grid points) as a terrain surface and triangulate it.
4. Calculate normal for each triangle.
5. Find normal at each grid point as weighed average of normals of all surrounding triangles. Normalize this normal.

4.3.2 Storing perturbed normals as normal map

Perturbed normals are stored in an image that is called normal map. Each normal is encoded as pixel colour based on the following formula-

$$R = \frac{Normal.x+1}{2}; \qquad G = \frac{Normal.y+1}{2}; \qquad B = \frac{Normal.z+1}{2};$$

4.3.3 Using normal map

For calculating specular light colour at a specific fragment of face, normal map is sampled using texture coordinates of the fragment for getting the encoded perturbed normal in the form of texel colour. This texel colour is decoded for getting the actual perturbed normal. Decoding is done using following formulas-

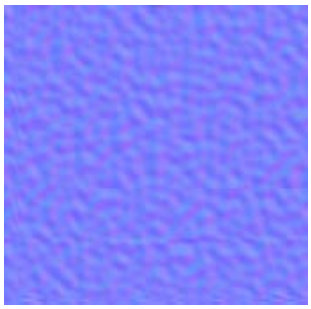


Figure 4.7: Normal map generated by Perlin noise.

$$Normal.x = 2 * R - 1; \qquad Normal.y = 2 * G - 1; \qquad Normal.z = 2 * B - 1;$$

Perturbed normal is used for calculating specular light colour. But this perturbed normal is in the image local coordinate system. We need to bring it in the coordinate system that is different for each triangle of the model. For this we need to have one transformation matrix (Tr) for transforming a normal from image local coordinates system to triangle local coordinate system. Tr is defined by three orthogonal vectors named as Bitangent (B), Tangent (T) and Normal (N) as shown below.

$$Tr = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

Calculation of B and T uses 3D positions of all three vertices of a triangle in model space and their corresponding 2D texture coordinates. Equation are given as following-

$$T = (\partial x/\partial u, \partial y/\partial u, \partial z/\partial u) = (-T_0, -T_1, -T_2)$$

OR

$$T_0 = \frac{[(v_1 - v_0)(x_2 - x_0) - (x_1 - x_0)(v_2 - v_0)]}{[(u_1 - u_0)(v_2 - v_0) - (v_1 - v_0)(u_2 - u_0)]}$$

$$T_1 = \frac{[(v_1 - v_0)(y_2 - y_0) - (y_1 - y_0)(v_2 - v_0)]}{[(u_1 - u_0)(v_2 - v_0) - (v_1 - v_0)(u_2 - u_0)]}$$

$$T_2 = \frac{[(v_1 - v_0)(z_2 - z_0) - (z_1 - z_0)(v_2 - v_0)]}{[(u_1 - u_0)(v_2 - v_0) - (v_1 - v_0)(u_2 - u_0)]}$$

and

$$B = (\partial x/\partial v, \partial y/\partial v, \partial z/\partial v) = (-B_0, -B_1, -B_2)$$

OR

$$B_0 = \frac{[(x_1-x_0)(u_2-u_0)-(u_1-u_0)(x_2-x_0)]}{[(u_1-u_0)(v_2-v_0)-(v_1-v_0)(u_2-u_0)]}$$

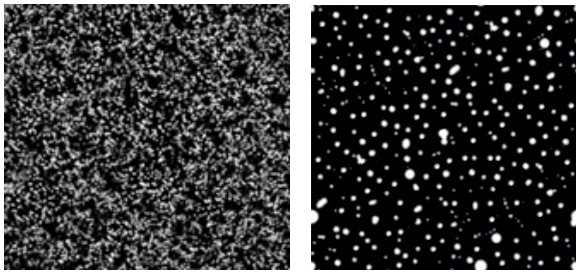
$$B_1 = \frac{[(y_1-y_0)(u_2-u_0)-(u_1-u_0)(y_2-y_0)]}{[(u_1-u_0)(v_2-v_0)-(v_1-v_0)(u_2-u_0)]}$$

$$B_2 = \frac{[(z_1-z_0)(u_2-u_0)-(u_1-u_0)(z_2-z_0)]}{[(u_1-u_0)(v_2-v_0)-(v_1-v_0)(u_2-u_0)]}$$

N can be calculated in two different ways. In one way, it will be cross product of T and B. In another way, it will be cross product of two edges of the triangle. We use second method for calculating N and then calculate B as cross product of T and N in fragment shader.

4.4 Gloss mapping

Normal mapping alone is not sufficient for giving the required effect. In order to achieve more realistic rendering we use gloss mapping. This is also an image file like texture with same dimensions. It acts like a mask for specular light calculated using perturbed normal. It is mostly black with some white spots distributed randomly across entire image. Specular light can only pass through if the sampled texel of gloss map is white for that fragment otherwise the specular light component will be 0 for that particular fragment. Two different gloss map is shown in Figure 4.8.



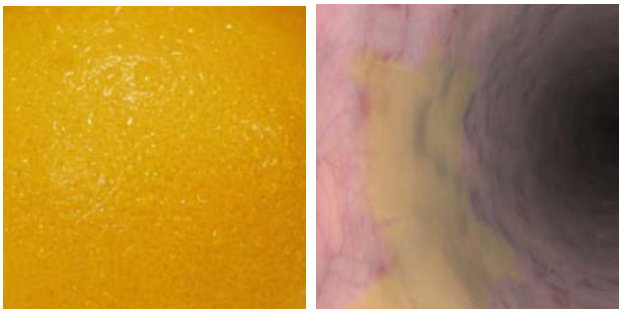
(a) (b)

Figure 4.8: Two different gloss maps.

4.5 Presence of Mucous

A patient is advised to not eat anything before endoscopy. So during the actual endoscopy, stomach and GI tract look very clean, but still some presence of mucous is visible. Although its presence is very less but where ever it is present, it appears very thick.

To approximate this mucous effect, we place it randomly throughout GI tract and stomach. We consider patches generated in Section 4.2 for mucous placement. For each patch we decide whether mucous should be placed on it or not. The decision is based on random event which answers yes with low probability. If decision comes in favour of mucous then we blend the colour of patch with color of mucous, where each colour has 50% contribution in final colour. Figure 4.9 shows presence of mucous in GI tract.

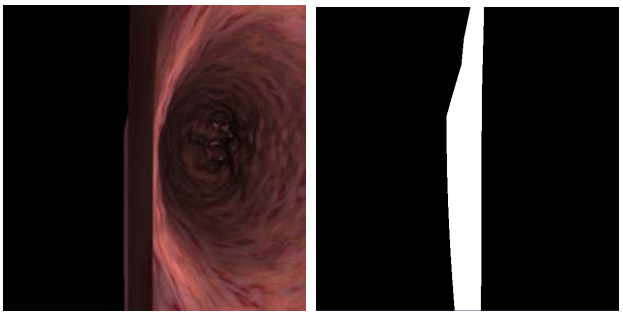


(a) (b)

Figure 4.9: Presence of mucous (a) Texture used for mucous. (b) Mucous in GI tract.

4.6 Collision Detection

We use power of GPU for collision detection. We write into two colour buffer in the fragment shader. First colour buffer is written with the original output colour. Second colour buffer is written by either black or white colour depending on the distance of fragment from the camera. If distance is less than some collision threshold distance then second colour buffer is written as black otherwise white. In the application program we read the second colour buffer from frame buffer and if any pixel is found black then we conclude that collision has occured and make the screen red otherwise we display the output of first colour buffer in the viewport. Figure 4.10 shows detection of collision if we go closer to GI tract wall.



(a) (b)

Figure 4.10: Collision detection (a) Full view of model and world before collision. (b) Full view of model and world after collision (wall is rendered as white).

4.7 Tube rendering

If the camera turns around then some portion of the endoscope tube should be visible, as shown in Figure 4.11. This is a rare situation but we need to take care of this feature for making simulated endoscopy session as real one.

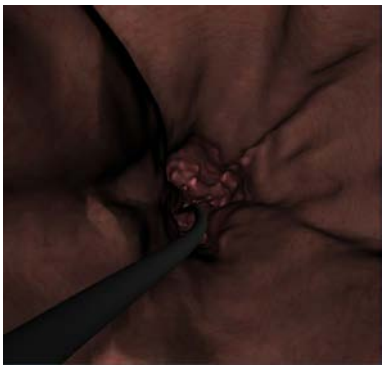


Figure 4.11: Endoscope tube rendering

As endoscope tube is a rigid tube, it can not have many bends. Camera and light source are mounted at the tip of endoscope tube. All the controls for turning around camera is available to doctor in form of knobs as shown in figure. Camera can be turned around by adjusting these knobs. Adjustment of knobs causes movement in some last few inches long portion of tube, lets call this portion as head of tube. We use this behaviour for tube rendering. Tube can be divided into two parts, head of tube and tail of the tube. We use B-spline curves for generating both parts of tube. As GI tract is a pipe like structure, some approximation of centerline can be found out and stored as pre-processing step in application program. We consider tail of tube as some portion of this centerline.

We use following procedure for centerline calculation -

1. Find all the triangles of GI tract which are cutting a plane parallel to $Z = 0$ plane. Find the center point by adding all such intersection points of triangles.

2. Find center points for all planes separated by some step value and covering entire GI tract.

3. Create centerline by calculating B-spline curve of all center points.

Head of tube is position dependent. If camera is in stomach then entire portion of tube inside stomach is free for movement, so it can be considered as head. If camera is in GI tract then every less portion of tube can move. In order to avoid unrealistic bend at joining point of two tube portions, we take last three points of tail into account while generating B-spline curve for head.

Following procedure is used for rendering of complete tube-

1. If camera is in GI tract then generate a cylinder along centerline of GI tract for tail of tube but till the third last center point before the current position of camera. Save the last three points.

If camera is in stomach generate a cylinder along centerline of GI tract for tail of tube but till the third last center point from the end of GI tract. Save the last three points.

2. Create a B-spline curve using three saved center points in GI tract and the current camera position. Generate another cylinder along this B-spline curve for head of tube.

4.8 Z-Line

There are two textures found in GI tract as we go inside. Surface where this change of texture is visible is called Z-line. We render this Z-line effect in fragment shader. We assume Z-line as a value of z coordinate. We render portion of mesh which is before the this value with one texture image and portion of mesh which is after the this value with another texture image. For smooth transition from one texture to another texture, we use blending of both the texture around the Z-line. Figure 4.12 shows the presence of Z-line.

4.9 Complete procedure for Graphical Interface rendering

Input: Triangular Mesh, two tissue textures, mucous textures, normal and gloss map.

Output: Rendering of mesh with wet surfaces.

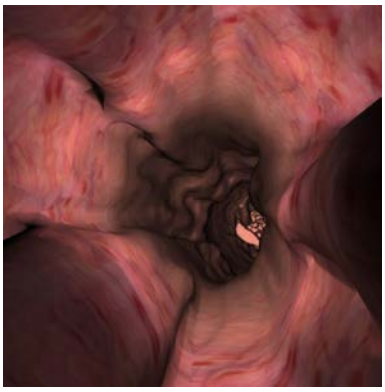


Figure 4.12: Z-line.

Processing Steps-

1. Process 3D mesh by following steps-
   (a) Extract each 3D vertex position P and indices of vertices of each triangular face available in the model.
   (b) Calculate normal for each triangular face.
   (c) Calculate N for each vertex by taking weighted average of all normals of triangles that shares the vertex. Weights are proportional to angle made by each triangle on the vertex.
   (d) Calculate texture coordinates (u, v) for each vertex by lapping method mentioned in section 4.2.
   (e) Calculate tangent for each triangular face as mentioned in section 4.3.
   (f) Calculate T for each vertex by taking average of all tangents of triangles that shares the same vertex.
2. Upload data on graphics hardware memory by following steps-
   (a) Set up ModelView and Projection matrices and upload them as uniform variables.
   (b) Upload P, N, T, (u, v) of each vertex using vertex buffer objects.
   (c) Upload light position and all images including texture images, mucous images, normal map and gloss map as uniform variables.
3. Start rendering pipeline by calling draw method. Following steps describe three stages of pipeline-

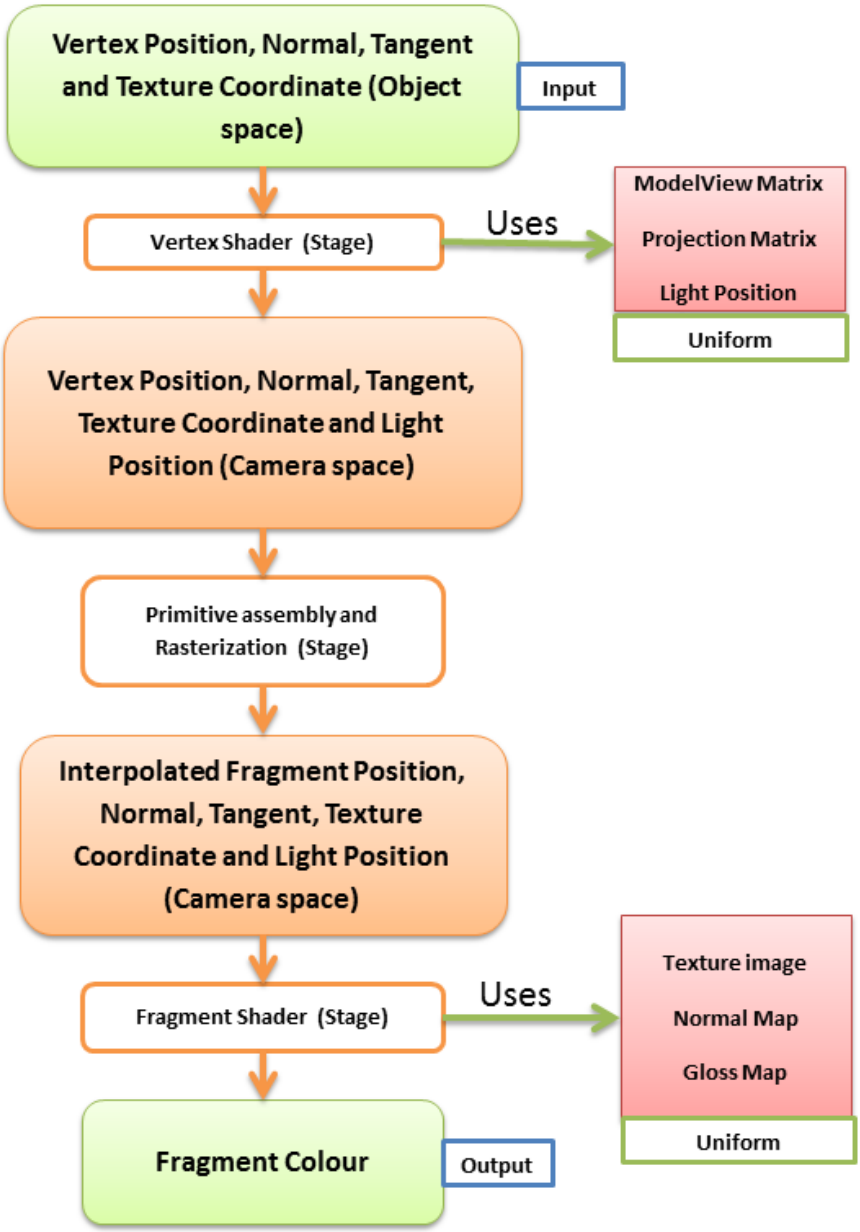


Figure 4.13: Programmable graphics pipeline

(a) Vertex Shader-

i. Input: P, N, T, (u, v) of each vertex. (All in Object space)

ii. Output: P, N, T, (u, v) and light position of each vertex. (All in camera space)

iii. Processing: Multiply incoming data by ModelView matrix for transforming it to camera space from object space.

(b) Primitive Assembly and Rasterizer-

i. Assemble all vertices that are part of a triangle.

ii. Find data for all fragments inside triangle by interpolating all data of three vertices. Data is nothing but incoming parameters (P, N, T, (u, v) and light position) from vertex shader for each vertex.

iii. Send interpolated data to fragment shader.

(c) Fragment Shader-

i. Input: Interpolated P, N, T, (u, v) and light position of each fragment. (All in camera space)

ii. Output: Fragment colour

iii. Processing-

A. Find all lighting vectors involved in lighting calculation.

L = normalize (light position - P).

V = normalize (-1 * P).

H = normalize (L + V).

N = normalize (N).

B. If length(P) < collision threshold, write black in second colour buffer otherwise white.

C. Calculate diffuse colour using N and L.

D. Find two components of T, one will be parallel to N and another will be perpendicular to N. Drop parallel component of T. Now N and T are perfectly orthogonal. Calculate B as cross product of N and T. Find Tr as T, B and N as basis vectors.

E. Sample perturbed normal from normal map. Convert it into camera space by multiplying it with Tr.

F. Find specular colour using transformed perturbed normal and H. Pass it through the gloss map. The resulting colour is the final specular highlight colour.

G. Write overall colour in first colour buffer as summation of diffuse colour, ambient colour and specular colour.

4. Read second colour buffer, if any pixel is black then display red colour in entire screen otherwise output first colour buffer followed by tube rendering.

4.10 Integration with mechanical system

Currently our graphical system is taking inputs from mouse and keyboard. Mouse and keyboard are used for moving camera which is mounted at the tip of endoscope tube. Figure 4.14 shows the proposed integration of graphical interface with mechanical system. After integration, our graphical system will be taking inputs from mechanical system and that will be nothing but camera position.

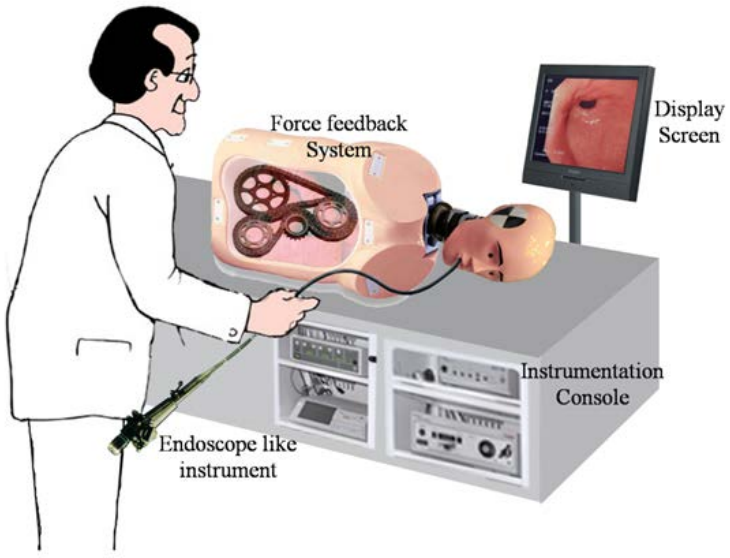


Figure 4.14: Integration (image source: http://cps.iisc.ernet.in/page/healthcareprojects)

Chapter 5

Optimizations for Performance Improvement

Performance is very crucial for any simulator. I have proposed some memory and time related optimizations for improving frame rate.

5.1 Memory Optimization

During texture mapping, we were overcoming straight line artifacts by using one extra set of texture coordinates and blending factor for bordering triangles. But in programmable pipeline, we send everything in the form of vertex attributes and this should be sent uniformly for all the triangles. So it does not matter whether a triangle is bordering one or non-bordering one. If we want to send one extra set of texture coordinates and blending factors then we must send these as vertex attributes and this will be for all the triangles. The advantage of this approach is, it needs just one pipeline execution for rendering one frame. But disadvantage would be wastage of GPU memory.

Another approach would be to divide all the triangle in two category, first will be bordering triangles and second will be non bordering triangles. In this case we will call pipeline two times for rendering one frame, first time for rendering first category triangles and second time for rendering second category triangles. Note that here we need to share the depth buffer in both the pipeline but the biggest advantage is to save significant amount of memory as first category vertices will have less attributes as compared to second category vertices. This approach will improve performance in low end graphics card having less memory.

One more optimization in texturing would be to merge single size patch with biggest adjacent patch.

5.2 Time Optimization

In collision detection we were writing encoded distances in a colour buffer and application program was reading this colour buffer for detecting the collision. However reading the colour buffer by application program is nothing but memory transfer between GPU (which executes graphics pipeline) and CPU (where application program resides). Before rendering each frame we need to check for collision, means transfer of memory. Memory is proportion to size of view port, so memory transfer in order of mega bytes. This can create a huge bottleneck in performance. However we can use an important observation for optimization it. As according to openGL convention, camera is always at origin looking at $-Z$ direction. So the triangle which is in collision with camera, will be having its fragments at around origin. So instead of transferring entire colour buffer we can transfer 2×2 middle portion of colour buffer and can detect collision. This approach will improve performance drastically.

Chapter 6

Results

Application program and shaders are written in OpenGL 3.3 and GLSL 3.30.6 respectively. Results are taken on 8 core 2.0 GHz 2 CPU Xeon machine with 8 GB of RAM. Machine also has an nVidia GeForce 8800 GTX graphics hardware with 4 GB of video memory. We are able to get good rendering of stomach and GI tract at the speed of ∼40 FPS on resolution of 1280 × 1024.

We have taken many features of a real endoscopy session into account. Although stomach and GI tract have very complex behaviour like contraction and expansion which results in dynamic changes in the mesh but we have limited our work to a static mesh. Techniques we are using are providing good visual results.

Images in Figure 6.1 represent the rendering of internal stomach surface. Figure 6.1(a) is first texture use, Figure 6.1(b) is second texture used, Figure 6.1(c) is normal map used for all results, Figure 6.1(d) is first gloss map used, Figure 6.1(e) is second gloss map used.

Images in Figure 6.1(f), 6.1(g), 6.1(h) are generated using first texture, Where Figure 6.1(f) shows rendering with neither normal nor gloss map, Figure 6.1(g) shows with only normal map, Figure 6.1(h) shows with both normal and first gloss map. Same is true with Figure 6.1(i), 6.1(j), 6.1(k) but they use second texture.

Images in Figure 6.1(l) and 6.1(m) use second gloss map. In Figure 6.1(l), rendering is generated using $\eta = 24$ but in Figure 6.1(m), $\eta = 40$.

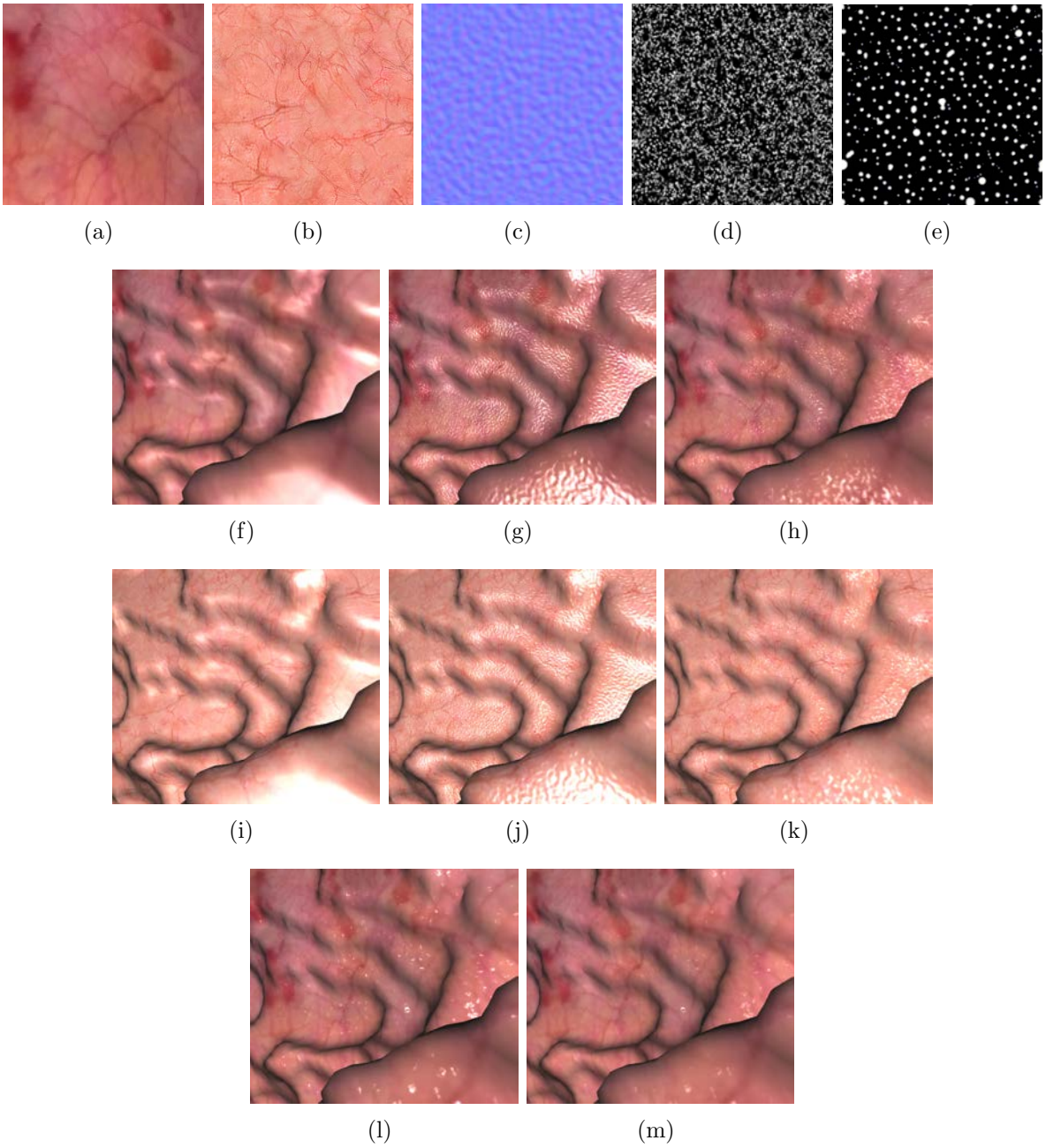


Figure 6.1: Results obtained on stomach surface (a) First texture. (b) Second texture. (c) Normal map used in all result images. (d) First gloss map. (e) Second gloss map. (f) Neither normal map nor gloss map (using first texture). (g) With only normal map (using first texture). (h) With both normal map and first gloss map (using first texture). (i) Neither normal map nor gloss map (using second texture). (j) With only normal map (using second texture). (k) With both normal map and first gloss map (using second texture). (l) With both normal map and second gloss map (using first texture where $\eta = 24$). (m) With both normal map and second gloss map (using first texture where $\eta = 40$).

Chapter 7

Conclusion

We have developed a graphical interface for an interactive endoscopy simulator. We have employed lapped texturing, normal mapping and gloss mapping for enhancing the visual appearance of tissue. We have also implemented important features like collision detection, tube rendering, mucous and Z-line.

Some possible improvements would be to use two levels of boundary triangles for blending in order to further overcome straight line artifacts. Merging of patches having more than one triangle with adjacent patch would further improve the performance. Tangent (T) and normal (N) may not be orthogonal at fragment level. The reason of this may be averaging involved during calculation of normal and tangent for each vertex and interpolation done by rasterizer. We drop parallel component of T in fragment shader for making it orthogonal to N. This may cause unexpected lighting for some fragments. Detailed analysis of this can be done as part of future work.

Appendix A

File Formats Used in Implementation

A.1 Object file format (.off)

These ASCII files are widely used to represent the geometry of a model by specifying the faces of the model's surface. OFF files alway begin with keyword OFF. The next line mentions the number of vertices, the number of faces, and the number of edges. The number of edges are always zero. The vertices are listed with x, y, z coordinates in object space, written one per line. After the list of vertices, the faces are listed, with one face per line. For each face, the number of vertices is specified, followed by indices into the list of vertices. Vertices are always numbered starting at 0. A simple example for a rectangle consisting of two triangles is given below:

```
OFF
4 2 0
-0.500000 -0.500000 0.000000
0.500000 -0.500000 0.000000
-0.500000 0.500000 0.000000
0.500000 0.500000 0.000000
3 0 1 2
3 0 1 3
```

A.2 Endoscopy simulator file format (.esff)

We have desined this file format based on our need. Unlike off format, here we store every vertex attribute in file. Basically we store complete vertex structure defined in our implementation. Calculation of all attributes takes huge time, which can be avoided using this file. This file can be generated offline. During some major deformation in mesh, we can avoid using this file

temporary. The structure of this file is same as off file with one change. Here one vertex will as many lines as number of attributes, where each line contains an attribute. Lines for all the faces will remain same. So j^th attribute of i^th vertex will be given by $i * n + j$ where n is total number of attributes and i and j start with 0. Above example with two attributes, position and normal, will look like below:

ESFF
4 2 0
-0.500000 -0.500000 0.000000
-7.500000 -3.500000 1.000000
0.500000 -0.500000 0.000000
.500000 3.500000 1.000000
-0.500000 0.500000 0.000000
4.500000 3.500000 4.500000
0.500000 0.500000 0.000000
7.500000 6.500000 20.000000
3 0 1 2
3 0 1 3

A.3 Dual graph file format (.dgff)

This is a very simple file format for storing dual graph (G) of a model (M). First line lists the number of nodes in G followed by equal number of lines for each node. Each line has three entries which show three edges of corresponding node in G.

A.4 GI tract center line file format (.gclff)

This is a very simple file format for storing center points of GI tract which are used in tube rendering. First line lists the number of center poinnts followed by center points themselves.

Appendix B

Important Data Structures Used in Implementation

B.1 Vertex

struct ver	
Member	**Description**
Vector3f pos	Position
Vector3f nor	Normal
Vector3f tan	Tangent
Vector2f uv_texture	Texture coordinate
Vector3f patch_col_texture	Colour of patch (for debugging)
float is_partial_patch	If vertex belongs to a border triangle of neighbouring patch
float partial_blending	Blending factor (valid only if is_partial_patch true)
Vector2f bound_uv_texture1	Another texture coordinates for blending (valid only if is_partial_patch true)
float is_mucous_present	If mucous present

B.2 Model

struct model	
Member	**Description**
ver *vers	List of vertices
model_triangle *triangles	List of faces
unsigned num_vert	Number of vertices
unsigned num_triangle	Number of faces

B.3 Dual Graph

struct graph	
Member	**Description**
model *mod	Model
int node_cnt	Number of nodes in graph
graph_node *nodes	List of node where every node has three edges

B.4 Structures for lapped texturing

B.4.1 Patch

struct patch_info	
Member	**Description**
int seed_trianlge	Seed trianlge
int triangle_count	Patch size
Vector3f patch_col	Colour of patch (for debugging)
float is_mucous_present	If mucous present on this patch

B.4.2 Duplicate Vertex

struct dup_ver_info	
Member	**Description**
int similar_ver	This vertex will share attributes of similar vertex
float u, v;	New texture coordinates
int which_triangle	This triangle will use this duplicated vertex instead of older one
int triangle_ver_num;	Which vertex of triangle out of three
Vector3f patch_col	Colour of patch (for debugging)
float is_mucous_present	If mucous present on this patch

B.4.3 Extending patch to one layer of neighbouring trianlgles

struct extend_patch_entry	
Member	**Description**
int tri	This triangle is bordering triangle
int partial_patch_num	Neighour patch of this trianle
Vector2f uv[3]	Other set of texture coordinates for three vertices
float blend[3]	Blending factors for three vertices

B.4.4 Class LapTexture

class LapTexture : Data member	
Member	**Description**
queue⟨int⟩ bfs_q	For doing BFS for patch growing
vector⟨dup_ver_info⟩ dup_ver	List of duplicated vertices
vector⟨int⟩ single_size_patch	List of patches having one triangle
vector⟨extend_patch_entry⟩ border_list	List of bordering triangles
int dup_ver_cnt	Total duplicate vertices count
patch_info *patches	List of all patches
int *isCovered	For keeping track of covered triangles is patches
int *random_array	For random seed selection
int *bfsDist	BFS distance of a triangle from seed triangle of correspond patch
int *patchNum	Current growing patch
int *covering_patch	Triagle covered by which patch
int *ver_share_cnt	How many triangles are sharing a vertex
int patch_size	Patch size of current patch
int patch_cnt	How many patches
int patch_freeq[100000]	For debugging
Vector3f available_patch_col[27000]	List of colours to be assigned to each patch
int seed	Seed for current patch
int cur_patch_size	Size of current patch
int cur_patch_num	Patch number of current patch
float rm[3][3]	Rotation matrix for current patch
Vector3f seed_normal	Normal of seed of current patch
int distortion	Distortion in current patch
int isCircular	If current patch circular
int circularDist	Circular radius of currnet patch

Appendix C

Framework

GPU gets the entire model during the starting of application. This model remains constant accross multiple execution of graphics pipeline. Uniforms remain constant during one execution of pipeline but later, can be updated usihg openGl calls from application program.

Entire framework of graphics system is shown in C.1 with internal workings. User interacts with system and changes camera and light positions. As a result, application program generates a new modelview matrix and it is sent to uniform section of graphics hardware along with light position. Application program triggers graphics pipeline. Execution of graphics pipeline gives two colour buffer, one for actual colour and another for collision detection. Application program checks for collision and red screen or original colour is displayed on the screen.

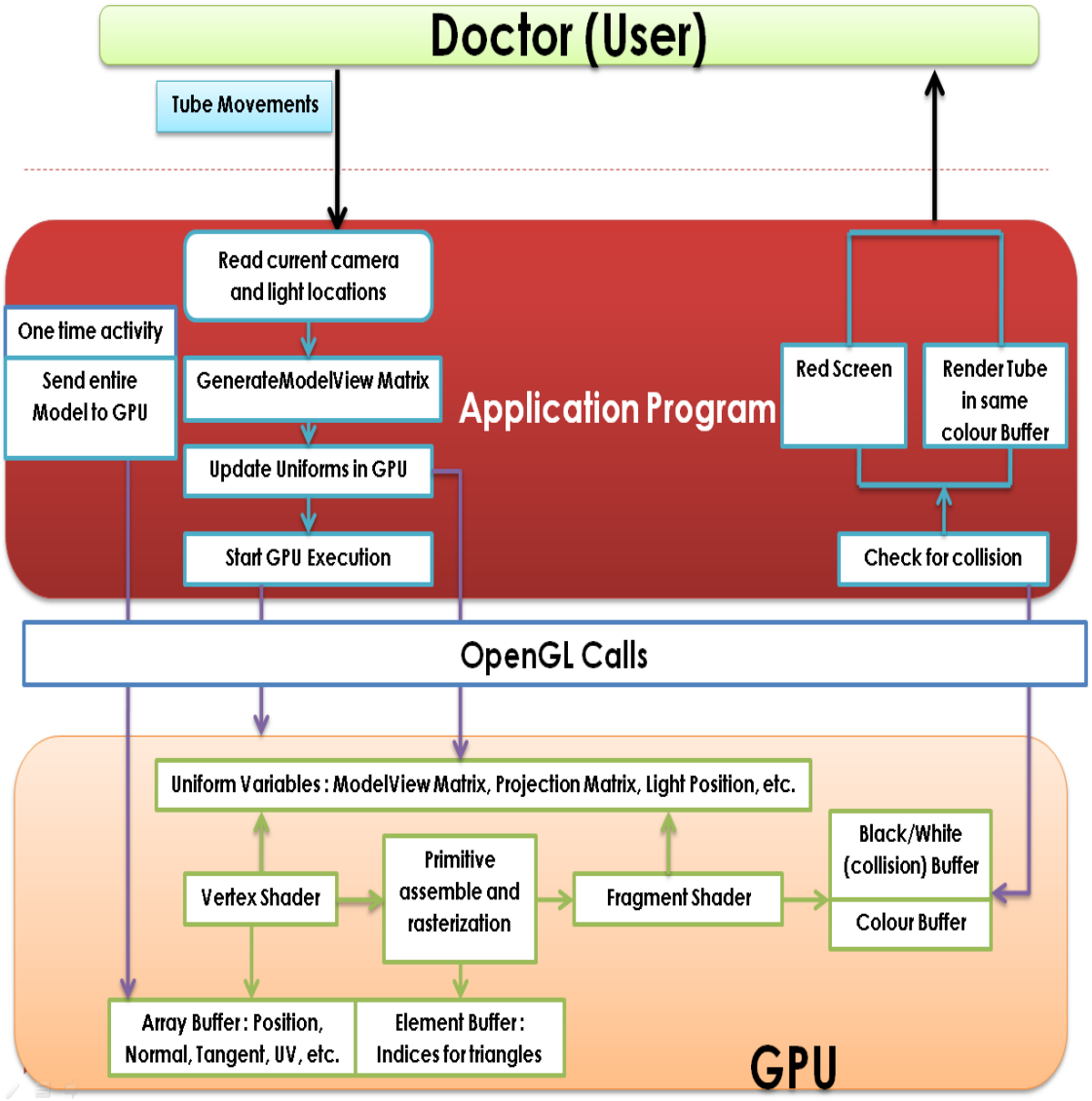


Figure C.1: Internal working of graphics system

Bibliography

[1] Glsl. https://www.opengl.org/documentation/glsl/. 6

[2] Off file format. http://segeval.cs.princeton.edu/public/off_format.html. 8

[3] opengl website. https://www.opengl.org/. 6

[4] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '77, 1977. 4

[5] M.A. ElHelw, B.P.L. Lo, A.Darzi, and G. Z. Yang. Real-time photo-realistic rendering for surgical simulations with graphics hardware. In *Medical Imaging and Augmented Reality: Second International Workshop, MIAR2004, Beijing, China, August 19-20, 2004. Proceedings*, pages 346–352, 2004. 3

[6] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, Vancouver, BC, Canada, 11 May 1992. 14

[7] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, 2002. 14

[8] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 465–470, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5. doi: 10.1145/344779.344987. URL http://dx.doi.org/10.1145/344779.344987. 3, 8

[9] T. Sangild Sørensen and J. Mosegaard. An introduction to GPU accelerated surgical simulation. In *Biomedical Simulation, Third International Symposium, ISBMS 2006*, pages 93–104, 2006. 2

[10] D. Stoyanov, M.A. ElHelw, B.P.L. Lo, A.J. Chung, F. Bello, and G.Z. Yang. Current issues of photorealistic rendering for virtual and augmented reality in minimally invasive surgery. In *Seventh International Conference on Information Visualization, IV 2003, 16-18 July 2003, London, UK*, pages 350–359, 2003. 3

[11] H.D. Visser, J. Passenger, D. Conlan, C. Russ, D. Hellier, M. Cheng, O. Acosta, Sébastien Ourselin, and Salvado. Developing a next generation colonoscopy simulator. *Int. J. Image Graphics*, 10(2):203–217, 2010. 3