CIS 636 Introduction to Computer Graphics

CIS 736 Computer Graphics

Spring 2011

Homework 3 (Problem Set) – Model Solution

Transformations and Viewing

## **Instructions and Notes (for the actual exam)**

* ~~You are permitted two (2) double-sided or four (4) single-sided, typewritten or handwritten pages of notes.~~
* ~~No calculators or computing devices are needed or permitted on this exam.~~
* You should have a total of 6 pages; ***write your name on each page***.
* There are **five (5)** problems. ~~You have 75 minutes for this exam. Budget your time carefully.~~
* Circle which course number (490, 490, or 636) you are enrolled under, both on this page and for each question, and answer the questions for that course number.

* ~~In the interest of fairness to all students, no questions shall be answered during the test concerning definitions.~~
* If you believe there is an error or ambiguity in any question, notify the instructor and ***state your assumptions***.
* You may use any consistent naming system for vectors and coordinate systems. However, if it does not match the OpenGL conventions or the systems used in Eberly or Foley *et al.*, then you are responsible for defining every vector by its full, unambiguous name.
* Use the space provided for your answers; you may add additional pages if needed.
* Select ***exactly one answer*** for each true/false and multiple choice question.
* Show your work on problems and proofs.
* There are a total of 100 possible points in this exam.

**Instructor Use Only**

1. \_\_\_\_\_\_ / 20
2. \_\_\_\_\_\_ / 20
3. \_\_\_\_\_\_ / 20
4. \_\_\_\_\_\_ / 20
5. \_\_\_\_\_\_ / 20

**Total \_\_\_\_\_\_ / 100**

1. **Definitions (5 parts, 4% each, 20% total).**
2. **Define: modelview transformation** Specify and define the coordinate systems involved. What coordinate systems does it map between?

The *modelview transformation* is the mapping from the coordinate system (*x*, *y*, *z*) of a scene component, the model, to the canonical world/scene coordinate system (*x*, *y*, *z*).

1. **(CIS 490 only) Define: parametric clipping** (and differentiate it from Cohen-Sutherland clipping, with outcodes and simultaneous equations)

*Clipping* is the problem of calculating what parts of a primitive object, such as a line segment or polygon, need to be rendered given a specification of the display region, the clip region.

Cohen-Sutherland clipping is a line segment clipping algorithm that pre-filters candidate line segments based on whether the endpoints both fall on the interior side of every clip edge or clip face (acceptance), and whether they both fall on the exterior of any face (quick rejection). Truncation requires finding the intersection if one is thus found to exist (in 2-D, solving a linear system of two simultaneous equations)

***Parametric clipping* calculates the point *t* where a line segment crosses a clip edge by solving for a dot product of 0 between a vector from a clip region corner to *t* and the edge/face normal. Such *t* are either where the line segment crosses in (potentially entering) or out (potentially leaving). Rejection of line sub-segments is done by taking the innermost PE/PL pair rather than using outcodes, and actual truncation of the segment is automatic.**

**(CIS 636 & 736) Define: culling** (give an example and differentiate it from clipping)

*Culling* in computer graphics means to eliminate objects or parts of objects from a scene that do not need to be rendered. Whereas clipping only eliminates the parts that do not fall inside the view volume, culling eliminates back faces and blocked parts of faces. View frustum culling solves one overlapping problem with clipping: the elimination of polygons that are strictly outside the view volume.

1. **Define: gluLookAt** (in terms of input, output, and function)

**gluLookAt()** is an OpenGL function that defines an arbitrary eye/camera space. It takes nine arguments: the world space (*x*, *y*, and *z*) coordinates of the Eye and At points and the direction of the Up vector, also given as an (*x*, *y*, *z*) vector.

1. **Define: fragment/pixel shading** (give one example and differentiate it from vertex shading in terms of the light model)

Vertex shading calculates properties of the vertices of a mesh, such as the lighting at each one. *Fragment shading* or *pixel shading* calculates properties of every pixel (element at a specific integer-precision coordinate in the normalized device coordinates or screen coordinates), often based on vertex shader output.

1. **(CIS 490 & 636) Define: texture mapping**

*Texture mapping* is the task of projecting a 2-D image onto a 3-D boundary representation in order to simulate the appearance of surface detail.

**(CIS 736 only) Define: bump mapping** (and explain briefly how it works)

*Bump mapping,* is the task of modifying the surface normals of a 3-D boundary representation (*e.g.*, a polygon mesh) so that it appears to have some desired surface detail such as a pattern of bumps or indentations. Lighting is calculated using the new normals, which are usually perturbed by adding values from a height map to the existing values. Thus, shadows and interobject reflection do not appear in bump mapping.

1. **Short Answer (20%)**
2. **(10%) Polygons-to-pixels pipeline.** Draw an illustration of the polygons to pixels pipeline, including the following:
   * camera coordinates / eye coordinates
   * model coordinates / object coordinates
   * normalized device coordinates (NDC)
   * screen coordinates
   * (optional) view coordinates / clip coordinates
   * world coordinates / scene coordinates

Specify the order in which Hproj, Hview, Hwindow, and Hworld go and what these have to do with the modelview transformation, normalizing transformation, perspective projection, perspective division, and viewport mapping. (**Note:** These are listed above in alphabetical order, not pipeline order!)

Also, explain exactly what coordinate systems the normalizing and viewing transformations map between and **how they are applied to vertices** **of the scene**.

**(See Eberly 2e § 2.3.2 – 2.3.7, pp. 48-66, especially p. 58)**

* 1. model coordinates / object coordinates Xmodel → (Hworld)
  2. world coordinates / scene coordinates Xworld → (Hview)
  3. camera coordinates / eye coordinates Xview → (Hproj)
  4. (optional) view coordinates / clip coordinates Xclip → (perspective division)
  5. normalized device coordinates (NDC) Xndc → (Hwindow)
  6. screen coordinates Xwindow

|  |  |
| --- | --- |
| Hworld: modelview transformation  Normalizing transformation: Xworld → Xndc { | Hview: “view matrix” **(really NT!)**  Hproj: projection matrix  /w: perspective division |
| Hwindow: window matrix  (*aka* viewport transformation) |  |

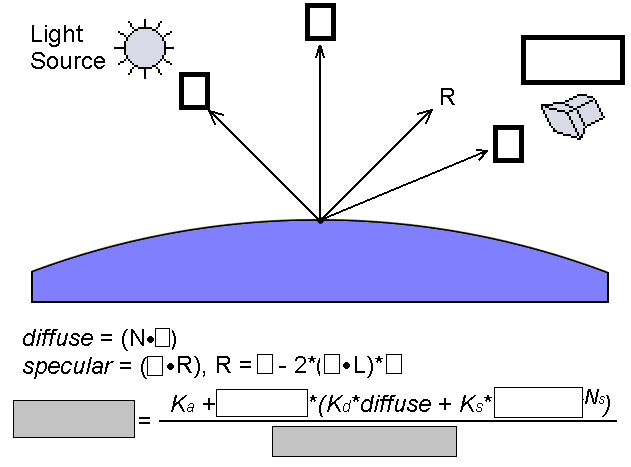
The above follows the terminology in Foley *et al.* (see vanDam’s notes from the intro graphics course at Brown University, CS123: <http://bit.ly/cDF6sL>). The idea is that to get the normalizing transformation that will take an arbitrary view (*r*, *u*, *d*) as input and transform vertices in our mesh correctly, we should calculate the viewing operations: “translate the origin to the eye point”, “rotate (*x*, *y*, *z*) into (*r*, *u*, *d*)”, *etc.* and **additively invert or transpose them**: “translate the eye point to the origin”, “rotate (*r*, *u*, *d*) into (*x*, *y*, *z*)”, *etc.* In other words, Hview maps the eye point to the origin and aligns and eye coordinate axes with world (scene) space axes, thus **normalizing** world space vertices.

**Note:** What Eberly calls Hview, the “view matrix”, is actually the *normalizing transformation* specified in Slides 23 – 31 of Viewing III (fall, 2009) in vanDam’s notes at Brown. To see this, see Slide 25: Eberly’s Hview has [***R***, ***U***, ***D***]T (row vectors). The actual viewing transformation is what Eberly refers to asHview-1, the “inverse view matrix” (and is the inverse of normalizing).

For further details and conventions specific to OpenGL, see:

<http://www.opengl.org/resources/faq/technical/transformations.htm>

1. **(10%) Types of mappings.** Define each of the following mappings in a brief sentence, illustrating as needed:
   * + - * **Bump mapping** is the task of modifying the surface normals of a 3-D boundary representation (*e.g.*, a polygon mesh) **for lighting**, to **simulate** surface detail.
         * **Displacement mapping** is the task of **deforming** a 3-D boundary representation (*e.g.*, a polygon mesh) so that it will **have** some surface detail.
         * **Reflection mapping** aka *environment mapping* is the task of projecting a surrounding scene onto a reflective object
         * **Shadow mapping** is the task of projecting the silhouettes of objects cast by light sources onto other objects
         * **Transparency mapping** is the task of simulating a see-through object by displaying the scene on the other side of it from the viewer, distorted by refraction, and blending colors based on translucency
2. **Illustrating Concepts (4 parts, 5% each, 20% total).** For the first two parts, fill in the boxes or blanks, and explain the filled-in value(s).
3. **(5%) 636 & 736: Phong illumination equation.**

****

Define the left-hand side that you wrote in the left gray box above (especially how it is used) and explain the denominator (what it represents).

Boxes, top to bottom and left to right:

*L*, *N*, *V*, camera/eye point

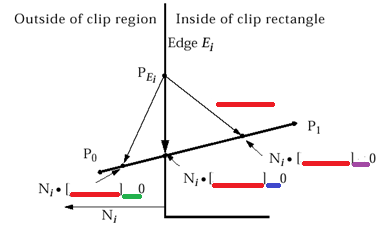
*L*

*V*, *L*, *N*, *N*

luminance, intensity, specular, *ad*2 *+ bd + c* (some references: *a0* + *a1r* + *a2r*2)

The left-hand side, *luminance*, denotes the intensity of (each) color displayed for a vertex of some object in the scene that is lit using the Phong illumination model. The denominator denotes the attenuation (fall-off of light energy) according to an inverse quadratic law, with linear and constant terms to allow more degrees of freedom than just inverse-square attenuation.

1. **(5%) 490 only: clipping, intersection calculation.** Fill in the blanks in the following figure and explain what the dot product’s relationship to 0 means. What does it mean to **solve for *t***? Explain what the variable D = P1 – P0 denotes and what cross product is in the denominator of the formula (mathematical expression) for *t*, and for what values it is meaningful.



Left to right:

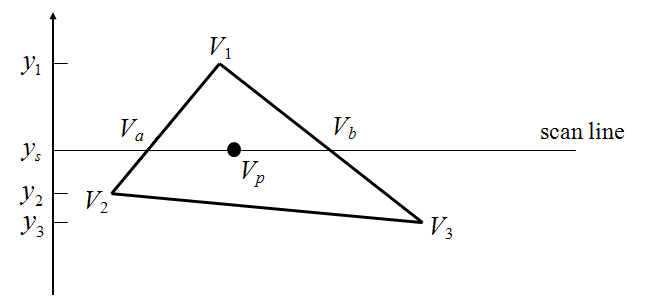
*P*(*t*) - , *>*

*P*(*t*)- , *=*

*P*(*t*)- , *<*

Solving for *t* means finding the parameter value for the clip point, where *P*(*t*)-  *=* 0 and the line segment crosses inside the clip region. = *P*1 - *P*0 denotes the vector from the one endpoint to another (used to check whether the line segment can cross the clip edge *Ei*: if • = 0, then and *Ei* are parallel and is trivially rejected).

**(5%) 636/736: scan line interpolation.** Consider the following generic illustration of scan line interpolation of value *v* over a triangle where *v1*, *v2*, and *v3* are known, along with *x1*, *y1*, *x2*, *y2*, *x3*, and *y3*. Give the formulas for the three values being interpolated on each line: *va*, *vb*, and *vp*. Finally, list two quantities that can be interpolated and indicate when this happens.

****

1. **(5%) 490/636: shading.** What is the advantage of Gouraud shading over Phong shading? What is a disadvantage?

*Gouraud shading* (aka *smooth shading*) is intensity interpolation using the Phong lighting model once per vertex. *Phong shading* is normal interpolation using the Phong lighting model once per pixel. Thus Gouraud shading tends to be much faster than Phong shading as the number of pixels per vertex goes up. When this happens, however, polygons get larger and the amount of expected error goes up because specular highlights can fall in the centers of polygons, far from a vertex. This leads to less realistic lighting of shiny objects.

.

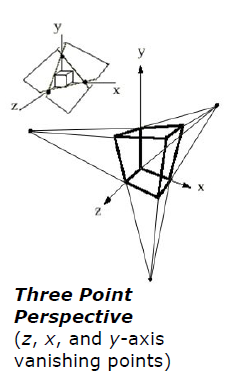
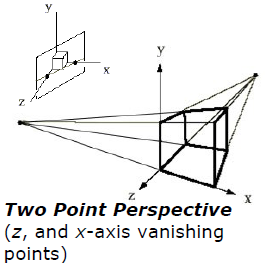
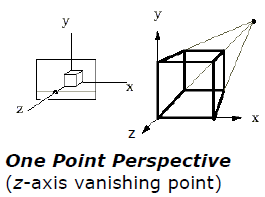
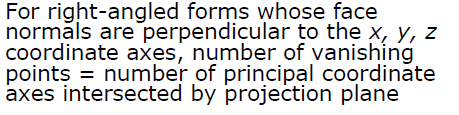
1. **(5%) 736 only: view specification.** What is the difference between a vertex shader and a pixel shader?

A *vertex shader* calculates properties of the vertices of a mesh, such as the lighting at each one. A *pixel shader* calculates properties of every pixel (element at a specific integer-precision coordinate in the normalized device coordinates or screen coordinates), often based on vertex shader output

1. **(5%) CIS 490 only: Bresenham’s midpoint line drawing algorithm**.How many symmetric cases are handled for line drawing, and why?

Eight (8). We can take the case of slope from 0 to 1 (45 degrees) and reflect across the 45-degree line, replacing “east” and “northeast” with “north” and “northeast”. We can call this “over” and “diagonal”. We can also handle symmetry across the *x* and *y* axes by using different logic for one endpoint “at the origin” and the other in the the first (up-right), second (up-left), third (down-left), or fourth (down-right) quadrant.

1. **(5%) CIS 636/736: perspective projections.** Explain and illustrate one, two, and three-point perspective.

****

1. **Fill in the blank (5 parts, 4% each, 20% total).**
2. Rotation and translation are examples of rigid-body transformations while shear is not.
3. (490/636) Translation is an affine transformation that is not linear.

(736) A surface that is Lambertian (white) has a high diffuse albedo.

1. Axonometric projections refer to all orthographic projections that are not aligned with one of the coordinate axes; a special case that is sometimes mistakenly used to refer to all such projections is called a(n) isometric projection.
2. *z*-buffering (*aka* depth buffering) is a(n) image-precision visible surface determination (VSD) technique.
3. Phong shading is a kind of shading that has to be implemented at the fragment level (e.g., using the OpenGL Shading Language) instead of using glShadeModel().
4. **Short answer/definition (4 parts, 5% each, 20% total).**
5. Define *rasterization* (*aka scan conversion*) and name some objects other than line segments that computer graphics libraries tend to provide tools for scan converting.

*Rasterization* is the problem of determining, given a model that is being viewed, what pixels of a screen should be turned on based on depth and occlusion. Triangles, quads, triangle fans, and triangle strips are often scan converted using hardware, while there is often software fallback for circles and ellipses.

1. Explain the difference between *illumination* and *shading*.

Illumination is the problem of calculating light intensity at vertices based on ambient, diffuse, and specular terms. Shading is a method for interpolating finding the color of a pixel based on illumination at either vertices or pixels.

1. What is the halfway vector *H* for a point light source? How and why is it used in the Phong illumination equation? Illustrate your answer here, referring to Problem 3a.

For normalized *L* and *V*, *H* = (*L* + *V*) / || *L* + *V* ||

N ⋅ H can be used instead of R ⋅ V, saving on the frequent recalculation as R and V change.

1. Consider the three arrays we discussed in lecture and which you used in homework to scan and represent a polygon mesh for an OpenGL function such as glDrawElements. Explain how the contents of these arrays are accessed based on the contents of others (and in what order they are organized).

.

Three arrays are needed:

* 1. int nvertices[] – an array of vertex counts per polygon
  2. GLfloat vertices[] – an array of world coordinates for all vertices, three at a time
  3. GLubyte indices[] – a flat array of vertex indices, with a number of elements equal to the sum of nvertices