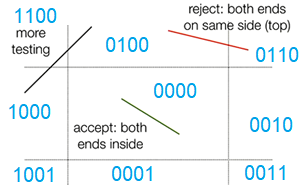
# 5. Meshes and Lighting

COMP3421 Computer Graphics • KC Notes

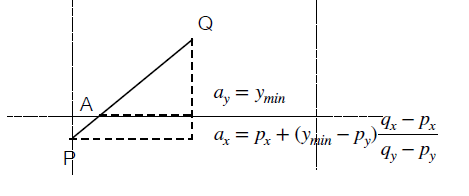
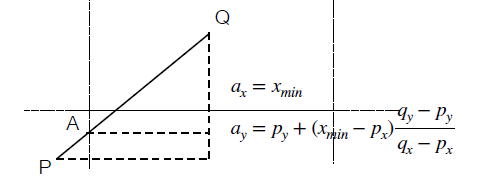
## 5.1 Clipping: Cohen-Sutherland

* Only render parts we can see
* **Cohen-Sutherland:** line clipped in an axis-aligned rectangle
  + Grid defined by bits – left, top, right, bottom, tells where point is in



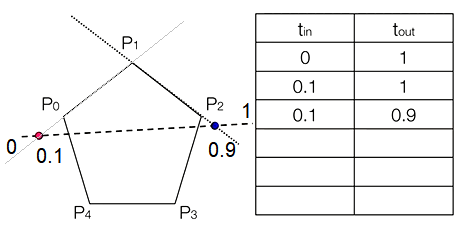
|  |  |
| --- | --- |
| Outcode(x, y):  code = 0;  if (x < left) code |= 8;  if (y > top) code |= 4;  if (x > right) code |= 2;  if (y < bottom) code |= 1;  return code; | |
| ClipOnce(px, py, qx, qy):  p = Outcode(px, py);  q = Outcode(qx, qy);  if (p == 0 && q == 0) {  // trivial accept  }  if (p & q != 0) { // both points in same grid location or all on one side  // trivial reject  }  if (p != 0) {  // p is outside, clip it  }  else {  // q is outside, clip it  } | |
| Clip(px, py, qx, qy): // clip until it fits in the rectangle  accept = false;  reject = false;  while (!accept && !reject):  ClipOnce(px, py, qx, qy) | |
|  |  |

* Clip using **similar triangles**, need to do twice as you don’t know where it crosses x/y
  + Maximum clip: 4 times, two on both sides.



## 5.2 Clipping: Cyrus-Beck

* **Cyrus-Beck** line clipped in a convex polygon
  + Figure out ray collisions
  + Ray: or
    - Check , at the collision:
* Normal is a vector perpendicular to a face of the polygon
  + If the ray is entering the polygon,
  + If the ray is exiting the polygon,

Steps:

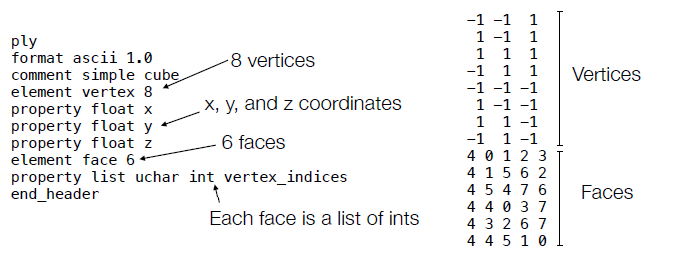
* Initialise (boundaries of the line)
* Compute **ray to each edge** of convex polygon
* Compute for each edge
* Keep track of the maximum and minimum

## 5.3 Meshes

* **Stitch polygons (usually triangle meshes) together** that forms skin of object
* Keep a list of **vertices and faces**
  + **Vertex list**: a list of points used in the mesh, given indices
  + **Face list**: lists of vertex indices – each face has the vertices as indices
    - In triangle mesh, each face has 3 vertices
* In OpenGL, have **2 buffers – transfer vertex buffer** and **element array buffer** with indices
  + vertexBuffer has items in buffer
  + indicesBuffer has items in buffer

|  |  |
| --- | --- |
| // In init (**transfer data once only**)  vertexBuffer = new Point3DBuffer(Arrays.asList(  new Point3D(-1,-1,1)...));  indicesBuffer = GLBuffers.newDirectIntBuffer(new int[] {  0,1,2, ...  });  int[] names = new int[2];  gl.glGenBuffers(2, names, 0);  verticesName = names[0];  indicesName = names[1];  gl.glBindBuffer(GL.GL\_ARRAY\_BUFFER, verticesName);  gl.glBufferData(GL.GL\_ARRAY\_BUFFER, vertexBuffer.capacity() \* 3 \* Float.BYTES,  vertexBuffer.getBuffer(), GL.GL\_STATIC\_DRAW);  gl.glBindBuffer(GL.GL\_ELEMENT\_ARRAY\_BUFFER, indicesName);  gl.glBufferData(GL.GL\_ELEMENT\_ARRAY\_BUFFER, indicesBuffer.capacity() \*   Integer.BYTES, indicesBuffer, GL.GL\_STATIC\_DRAW);  Shader.setModelMatrix(gl, frame.getMatrix());  gl.glDrawElements(GL.GL\_TRIANGLES, indicesBuffer.capacity(),  GL.GL\_UNSIGNED\_INT, 0);  gl.glDeleteBuffers(2, names, 0); | |
| // In draw function (**only update current buffer and draw**)  gl.glBindBuffer(GL.GL\_ARRAY\_BUFFER, verticesName);  gl.glVertexAttribPointer(Shader.POSITION, 3, GL.GL\_FLOAT, false, 0, 0);  gl.glBindBuffer(GL.GL\_ELEMENT\_ARRAY\_BUFFER, indicesName);  Shader.setModelMatrix(gl, frame.getMatrix());  gl.glDrawElements(GL.GL\_TRIANGLES, indicesBuffer.capacity(),  GL.GL\_UNSIGNED\_INT, 0); | |
|  |  |

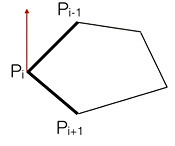
* PLY format: format of storing indices
  + Should be loaded in init

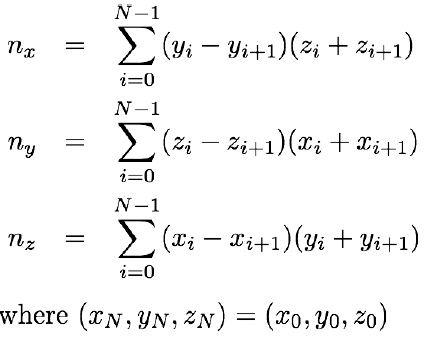


## 5.4 Lighting

* Achromatic Light: lighting with no colour, only intensity
* Local illumination: only model reflections directly from a light source
  + In real life, light reflects from one object to another but is very costly to model
* Three types of reflection
  + **Diffuse** **reflection**: light hits a surface and goes in all directions
    - Matte/dull surfaces exhibit diffuse reflection, does not depend on viewport
  + **Specular reflection**: light reflected at the same angle
    - Polished surfaces, look different from different view points
  + **Ambient** light covers lighting from indirect surfaces, does not depend on light/viewer position
* Lighting equation – intensity:
* Three important vectors:
  + **Normal vector** : vector perpendicular to the surface at point P
  + **View vector** : vector from point P to camera
  + **Source vector** : vector from point P to the light source

## 5.5 Lighting: Normals

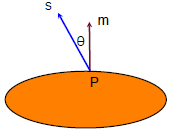
* On flat surfaces, use **face normals** – set normal perpendicular to the face
  + Find cross product of 2 sides if polygon is convex
  + Make sure it is , because anticlockwise
  + Normalise the normals
  + **Newell’s method** for arbitrary polygons



* On curved surfaces, use **vertex normals**, specify different value that changes more gradually
  + Mathematically, take derivative and take normal
  + Take average of surrounding face normals by adding all together

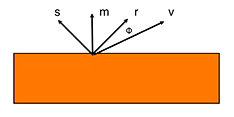
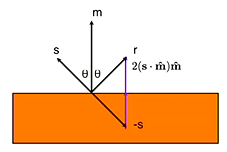
|  |  |
| --- | --- |
| // Assuming **non-indexed**, calculating face normals:  private void computeFaceNormals() {  for (int i = 0; i < vertices.capacity() / 3; i++) {  Vector3 n = normal(vertices.get(i\*3), vertices.get(i\*3 + 1),  vertices.get(i\*3 + 2));  normals.put(i\*3, n.asPoint3D());  normals.put(i\*3 + 1, n.asPoint3D());  normals.put(i\*3 + 2, n.asPoint3D());  }  } | |
| // Assuming non-indexed, calculating vertex normals, normals are not normalised  private void computeVertexNormals() {  // Initialise the normals to the zero vector  for (int i = 0; i < normals.capacity(); i++) {  normals.put(i, 0, 0, 0);  }    // Add the face normals of all surrounding faces.  for (int i = 0; i < indices.capacity() / 3; i++) {  int index1 = indices.get(i\*3);  int index2 = indices.get(i\*3 + 1);  int index3 = indices.get(i\*3 + 2);    Point3D p1 = vertices.get(index1);  Point3D p2 = vertices.get(index2);  Point3D p3 = vertices.get(index3);    Vector3 normal = normal(p1, p2, p3);  normals.put(index1, normals.get(index1).translate(normal));  normals.put(index2, normals.get(index2).translate(normal));  normals.put(index3, normals.get(index3).translate(normal));  }  } | |
|  |  |

## 5.6 Diffuse scattering

* **Diffuse scattering**: equal in all directions so does not depend on viewing angle
  + Amount of reflected light depends on angle of source of light
    - Small angle = smaller lit area, higher intensity
    - Large angle = larger lit area, lower intensity
* Lambert’s Cosine Law
  + Where is the source vector and the normal vector, both **normalised**,   
     source intensity, the diffuse reflection coefficient
  + When angle is 0, all light reflected, when angle is 90, we get 0 intensity
* to prevent **negative cosine**
* Coefficient:
  + Light surfaces have values close to 1, reflect more light

## 5.7 Specular reflection

* Specular reflection: **some scattering** of light
  + Done via Phong model
* projects s onto m
* Take **angle between view and reflected vector** to obtain intensity
* Phong exponent – as it increases, it makes smaller – less scattering and more mirror-like



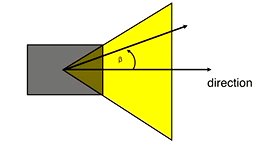
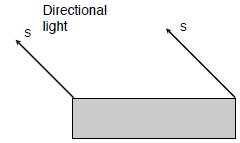
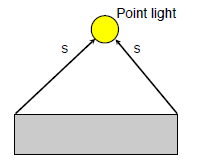
* **Blinn Phong model** uses a vector **halfway between source and viewer** instead of calculating the reflection vector

## 5.8 Ambient light

* Add ambient light level to scene:
  + With the ambient light intensity, the ambient reflection coefficient
  + Usually

## 5.9 Combining lights and light limitations

* **Combine light contributions** for a particular light source by adding
* **Combine all lights** by adding all components of all lights together
* Limitations:
  + Only a local model – shiny objects do not reflect light, only light source
  + Colour at each vector depends only light properties and material properties
  + Does not take into account shadows, vertices being obscured
* Two types of light **point and directional**
  + Point lights **come from a point in the world** – source vector is from surface to point light
  + Directional lights **comes along a particular direction** – source vector is the same everywhere
* Moving lights need to be **subjected to the same modelling transformation** as the object
* Spotlights have a **direction and cutoff angle**, check if source is within an angle
  + Attenuation – brightness falls off as you move from centre ( the attenuation factor)



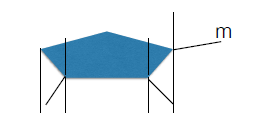
## 5.10 Shading

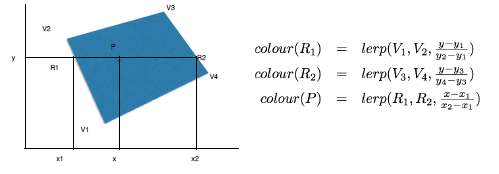
* When do we apply the lighting?
  + **Flat shading**: calculate for each face
  + **Gouraud shading**: calculate for each vertex and interpolate for every fragment
  + **Phong shading**: calculate for every fragment

## 5.11 Flat shading

* Shade the **entire face the same colour**
  + Compute intensity for some point on the face, e.g. the first vertex
  + Set every pixel to that value
* Good for diffuse illumination, flat surfaces, distant light sources, non-realistic/retro rendering, fastest shading

## 5.12 Gouraud shading

* Lighting equations are calculated for each **vertex using an associated vertex normal**
  + Normal m is accessed from buffer with same size as vertex buffer
* Smooth out rough edges by **tweaking normals** at edges
  + Normals for every fragment are done by **interpolating on neighbouring vertices**



|  |  |
| --- | --- |
| gl.glBindBuffer(GL.GL\_ARRAY\_BUFFER, normalsName);  gl.glVertexAttribPointer(Shader.NORMAL, 3, GL.GL\_FLOAT, false, 0, 0); | |
|  |  |

* Good for curved surfaces, close light sources, diffuse shading, more expensive
  + Handles specular highlights poorly – works if highlight occurs at a vertex, but might appear to jump from vertex to vertex

## 5.13 Phong shading

* Defers the illumination calculation until **fragment shading step** (fragment shader)
* Calculated per fragment rather than per vertex
  + Need to move over **view position** and intensity/material properties
  + Interpolate **source and normal vectors** (out them in vertex shader)
  + Interpolating normal/source/view vectors (instead of illumination)
* Good: Handles specular lighting well, improves diffuse shading, more physically accurate
* Bad: Slower than Gouraud as normals and illumination values need to be calculated per pixel rather than per vertex

## 5.14 Lighting Summary

* **Normal vector** : vector perpendicular to the surface at point P
* **View vector** : vector from point P to camera
* **Source vector** : vector from point P to the light source

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Relative position of object to…** | | **Equation** |
| Light | Viewer |
| **Ambient** | No | No |  |
| **Diffuse** | Yes | No |  |
| **Specular** | Yes | Yes |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Shading** | **Where** | **Good for** | **Bad for** |
| **Flat** | Face – each face has a uniform colour | Efficiency, flat surfaces, retro/blocky look | Curved surfaces, specular highlights |
| **Gouraud** | Vertex – intensity at every vertex, interpolate intensity between vertices | Curved surfaces, diffuse shading | Specular highlights, doesn’t have specular dot at the front, specular jumps around |
| **Phong** | Fragment | Handles specular lighting well, improves diffuse shading, more physically accurate | Old hardware |