Computer Graphics

(Curves and Surfaces, Part 2)

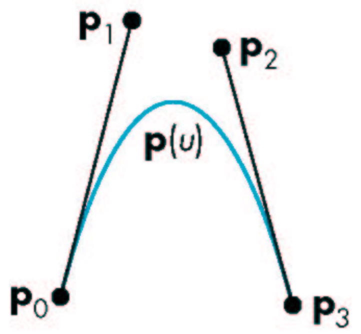
Thilo Kielmann
Fall 2003
Vrije Universiteit, Amsterdam
kielmann@cs.vu.nl

http://www.cs.vu.nl/~graphics/

Outline for today

- NURBS
- Rendering splines
- The Exam

Bezier curves



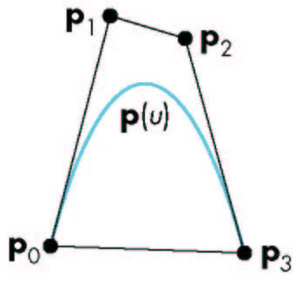
Use p_0 and p_3 for interpolation
Use p_1 and p_2 to approximate tangents
(like with Hermite curves)

Joins of Bezier curves

We use again $p_0 \dots p_3$ for the first curve and $p_3 \dots p_6$ for the second.

Bezier curves are C^0 continuous, but **not** C^1

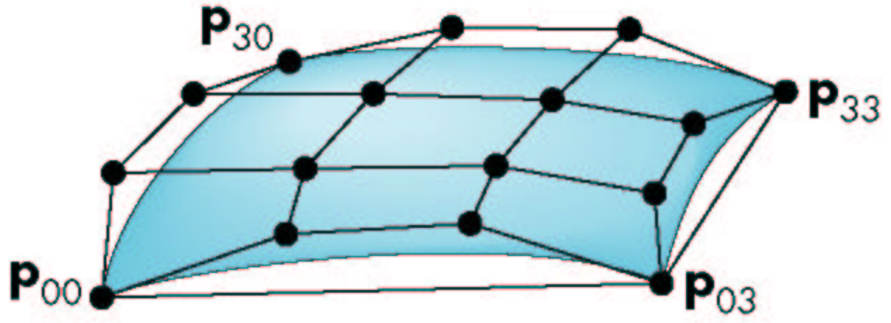
Bezier polynomial and its convex hull



The Bezier polynomial in terms of its blending polynomials forms a convex sum. (property of the blending polynomials).

Thus the Bezier polynomial lies within the convex hull of its control points, which is close to by not exactly interpolating all control points.

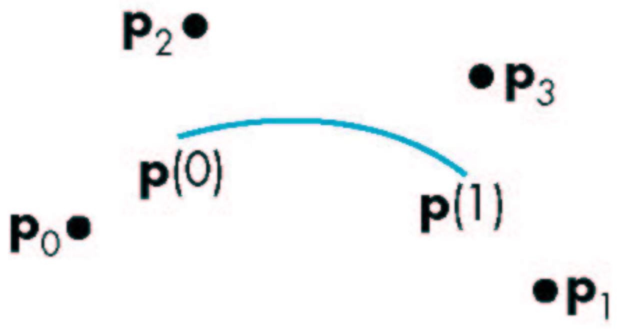
Bezier surface patch



completely contained in the convex hull of its control points

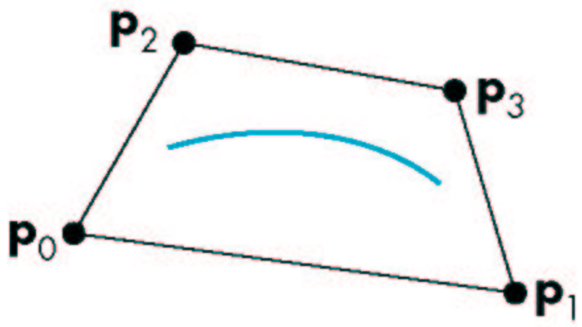
interpolates $p_{00}, p_{03}, p_{30}, p_{33}$

Cubic B-Splines



To improve smoothness, give up interpolation completely. Let control points define curve only between middle control points.

Convex hull for B-spline curve



Again, control points define convex hull for the curve.

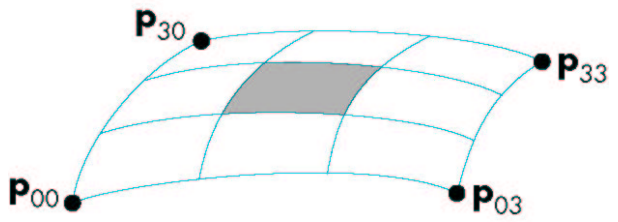
Continuity of B-Splines

B-Splines have been designed to be C^0 and C^1 continuous.

They happen to be also C^2 continuous.

This is very smooth, like blending metal.

Spline surface patch



from the blending functions:

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u)b_j(v)p_{ij}$$

defines only the middle patch of the surface

General B-Splines

Set of control points $p_0, \ldots, p_m$
approximation problem: find $p(u) = [x(u) \quad y(u) \quad z(u)]^T$
over $u_{\text{min}} \le u \le u_{\text{max}}$

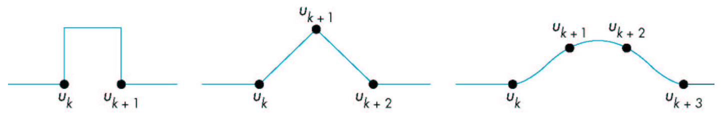
Set of values $\{u_k\}$ called **knot array**
$u_{\text{min}} = u_0 \le u_1 \le \ldots \le u_n = u_{\text{max}}$

$$p(u) = \sum_{j=0}^{d} c_{jk}u^j, \quad u_k < u < u_{k+1}$$

(for $d = 3$ and n knots we have to solve $4n$ equations)

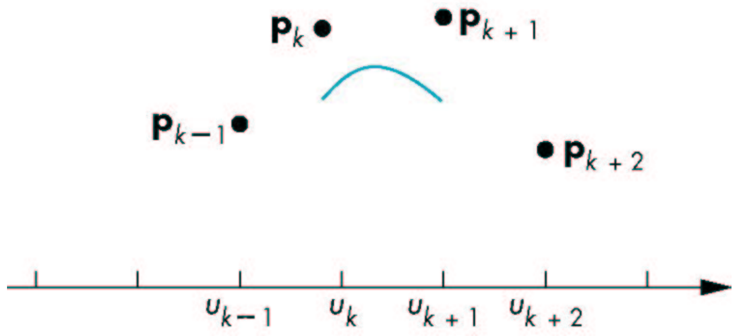
Recursively defined B-Splines

Local solution, by combining basis functions for each interval



These **Basis splines** ($\rightarrow$ name B-splines) of degree d are non-zero only in $d+1$ intervals.
(and they are recursively defined, needs $d-1$ extra knots for the ends)
C^{d-1} continuity at the knots

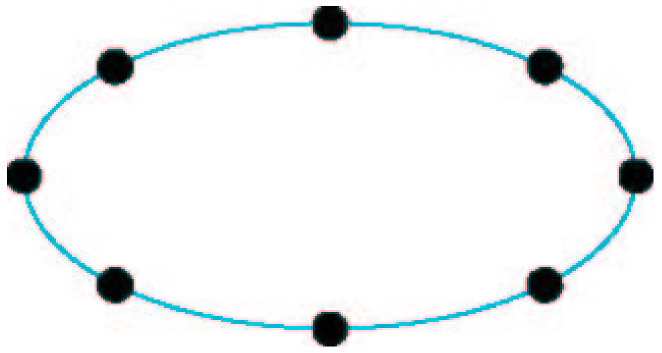
Uniform B-Splines



Splines are called **uniform** if their knots are equally spaced.

Also non-uniform spacing and repeated knots possible.

Periodic uniform spline



(example, periodicity depends on knot array)

Open splines

If a knot has a multiplicity $d + 1$, a spline of degree d must interpolate it. (We can use this to define the endpoints of a spline.)

Example: $\{0, 0, 0, 0, 1, 2, \ldots, n-1, n, n, n, n\}$

$\{0, 0, 0, 0, 1, 1, 1, 1\}$ is the cubic Bezier curve

NURBS

B-Splines work in 3D, in 2D, and in 4D
$\rightarrow$ homogeneous coordinates

control point $p_i = [x_i \;\; y_i \;\; z_i]^T$

weighted homogeneous representation: $q_i = w_i \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}$

Weights determine the relative importance of a control point.

$$p(u) = [x(u)\ y(u)\ z(u)\ w(u)]^T$$

$$q(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} = \sum_{i=0}^{n} B_{i,d}(u) w_i p_i$$

$$w(u) = \sum_{i=0}^{n} B_{i,d}(u) w_i$$

$$p(u) = \frac{1}{w(u)} q(u) = \frac{\sum_{i=0}^{n} B_{i,d}(u) w_i p_i}{\sum_{i=0}^{n} B_{i,d}(u) w_i}$$

Nonuniform, Rational B-Spline (NURBS)

- nonuniform (no assumptions about knots)
- rational function, with “built-in perspective division”
- NURBS look better in perspective views (perspective is no affine transformation)
- quadrics can be rendered as special case of NURBS (uniform mechanism)

Rendering polynomials

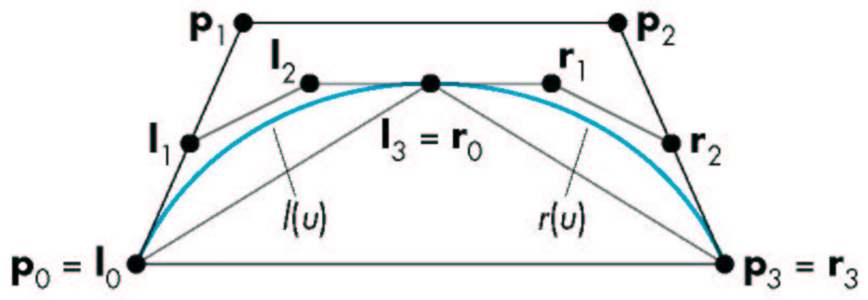
Horner’s method:

$p(u) = c_0 + u(c_1 + u(c_2 + uc_3)))$

takes $n = 3$ multiplications

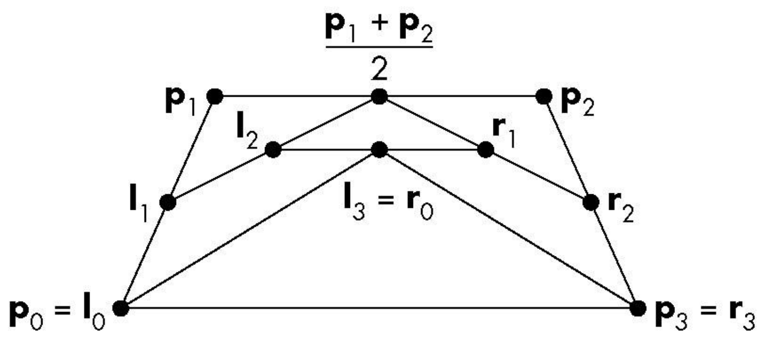
The “forward differences” methods only needs additions, but is sensitive to numerical error propagation.

Recursive Subdivision (Bezier)

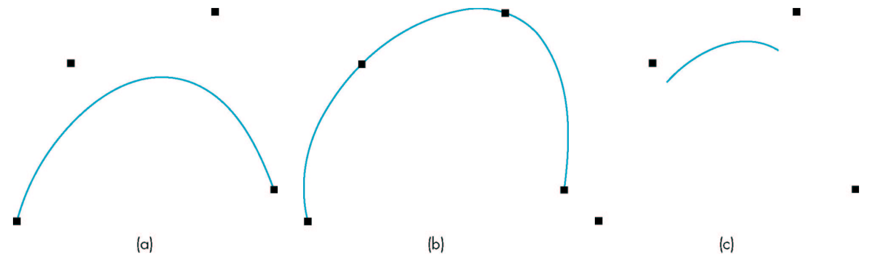


Divide recursively until convex hull gets close to straight line.
Best done in screen coordinates / resolution determines recursion depth.

Construction of Subdivision Curves


$\frac{\mathbf{p}_1 + \mathbf{p}_2}{2}$
$\mathbf{p}_1$
$\mathbf{p}_2$
$\mathbf{l}_2$
$\mathbf{r}_1$
$\mathbf{l}_1$
$\mathbf{l}_3 = \mathbf{r}_0$
$\mathbf{r}_2$
$\mathbf{p}_0 = \mathbf{l}_0$
$\mathbf{p}_3 = \mathbf{r}_3$

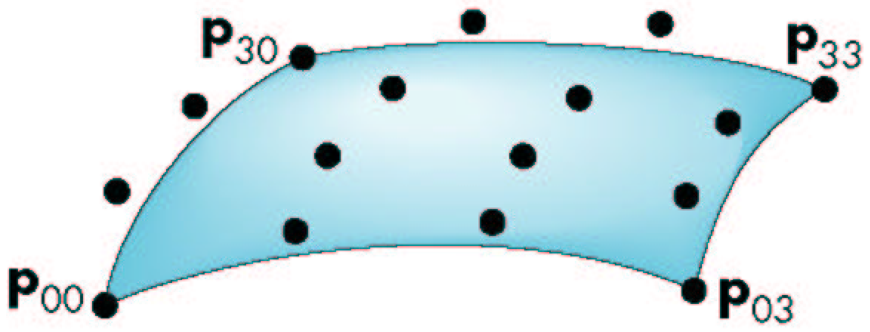
Rendering Other Polynomial Curves



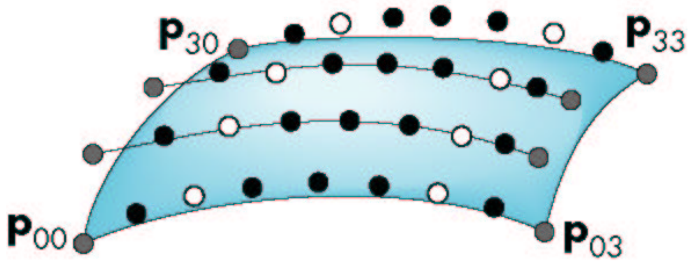
Bezier, interpolation, and B-spline polynomials can be seen as different representations ofthe same curve.

After transformation (matrix) to Bezier polynomial, do recursive subdivision.

Subdivision of Bezier Surfaces

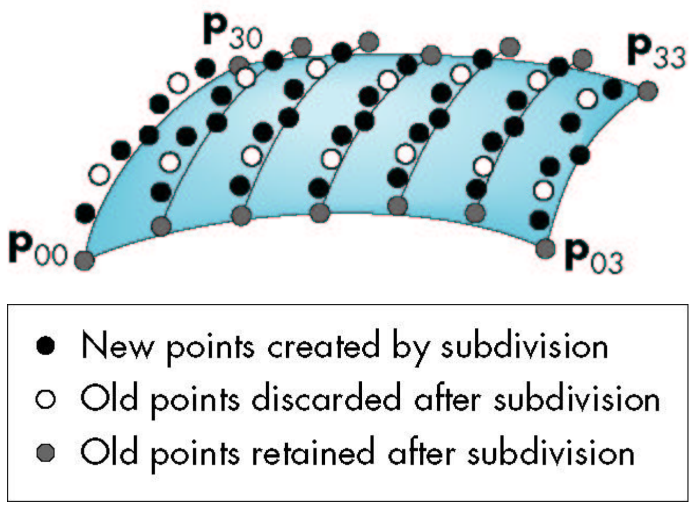


First Subdivision

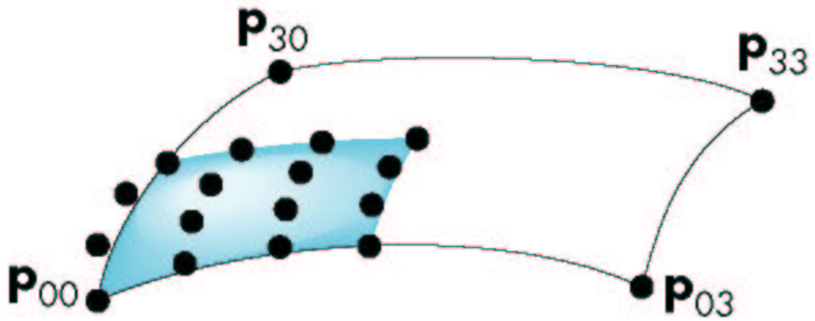


- ● New points created by subdivision
- ○ Old points discarded after subdivision
- ◍ Old points retained after subdivision

Second Subdivision

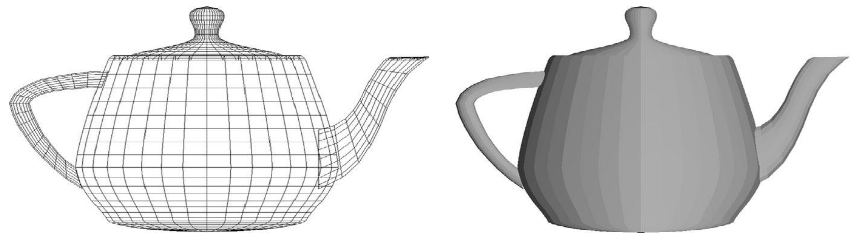

P_{30}
P_{33}
P_{00}
P_{03}
● New points created by subdivision
○ Old points discarded after subdivision
● Old points retained after subdivision

A Subdivided Quadrant



Flatness test (recursion end) hard to do, mostly left to the programmer.

The Utah Teapot



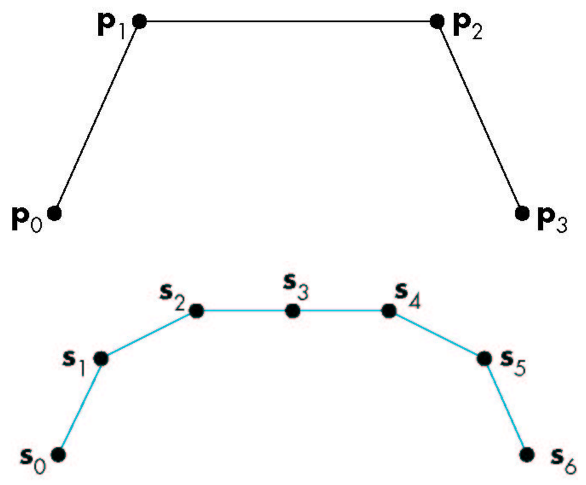
Created by M.Newell at Univ. of Utah, in the 1970s

32 bicubic Bezier patches from 306 control points

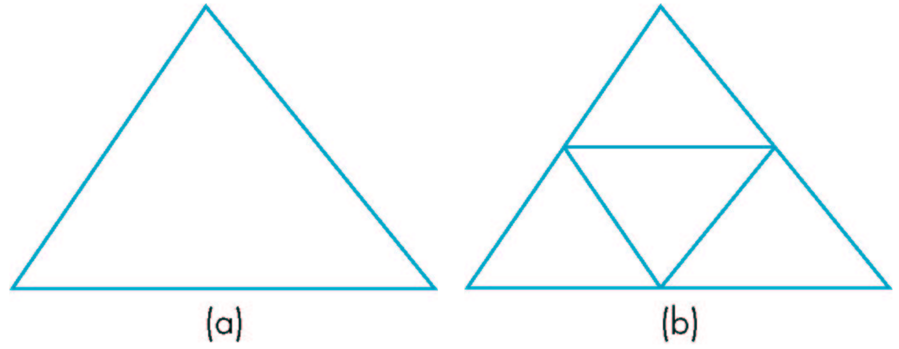
Rendering Quadrics

- polynomials of the form $x^i y^j z^k$, with $i + j + k \leq 2$
- can be rendered by raycasting (solving quadratic equation)
- can be rendered by subdivision

Subdivision of Piecewise, Linear Curve

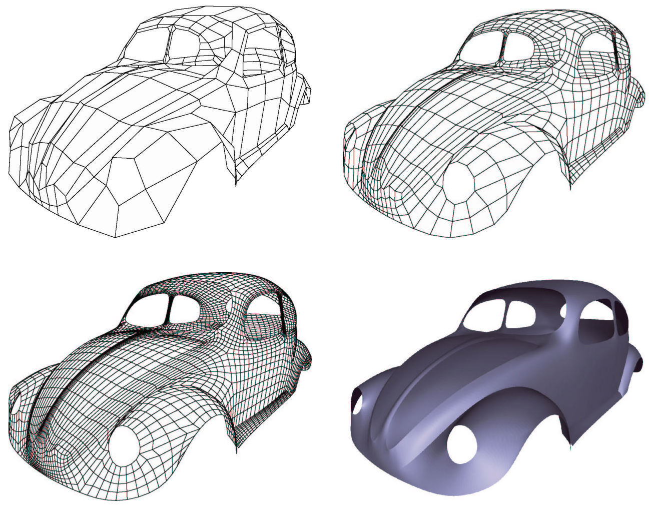


Triangle subdivision



subdivided surfaces are a refinement of the original surface

Subdivision of Polygonal Mesh



Curves and Surfaces in OpenGL

- using **evaluators** for generating Bezier polynomials
- 1D up to 4D
- 1D = curves
- 2D = surfaces

Bezier Curves

Define and use a 1D evaluator:

```
glMap1f(type, u_min, u_max, stride, order, point_array)
// type = GL_MAP1_VERTEX_3, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1...
// stride e.g. 3 for 3 points per curve segment
// order = degree+1
glEnable(type);


glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, data);
glBegin(GL_LINE_STRIP);
    for (i=0; i<100; i++) glEvalCoord1f( (float) i/100.0);
glEnd();
```

Bezier Surfaces

Define and use a 2D evaluator:

```
glMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4, 0.0, 1.0, 12, 4, data);
glEnable(GL_MAP2_VERTEX_3);


for (j=0; j<100; j++){
  glBegin(GL_LINE_STRIP);
     for (i=0; i<100; i++)
         glEvalCoord2f( (float) i/100.0), (float) j/100.0);
  glEnd();
  glBegin(GL_LINE_STRIP);
     for (i=0; i<100; i++)
         glEvalCoord2f( (float) j/100.0), (float) i/100.0);
  glEnd();
}
```

Quadrics in GLU

```
GLUquadricObj *p = gluNewQuadric();
gluQuadricDrawStyle(p, GLU_LINE);

// from the picking robot
gluCylinder(p, BASE_RADIUS, BASE_RADIUS, BASE_HEIGHT, 5 ,5);
// 5 and 5 are the subdivision slices in x and y

also:
gluDisk
gluPartialDisk
gluSphere
```

Summary Curves

- Bezier and B-Spline polynomials
- Rendering by recursive subdivision
- GLU provides some useful quadrics objects

Written Exam

- January 19, 2003 (13:30–16:30), M1.29
- Registration via TISVU
- Second Chance ("Herkansing"):
  June 18, 2003 (13:30–16:30), S2.09
  (with only few participants:
  oral exam, same time, same place)
- Check the announcements of the "onderwijsbureau"
  for changes of time and/or place. . .

Written Exam (2)

- New: "closed book exam"
- No materials (books, slides, etc) allowed

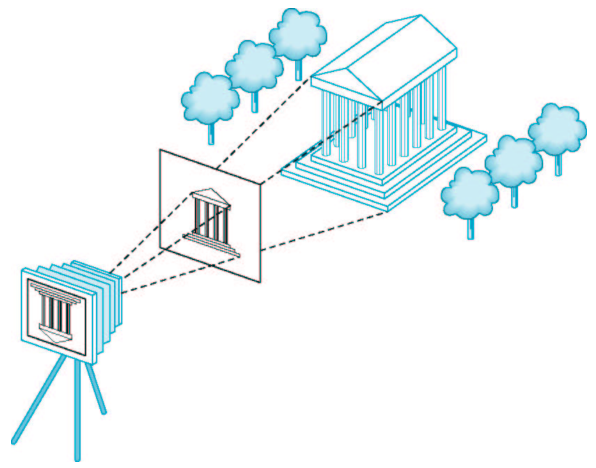
Written Exam (3)

- Topics: Intersection between book (**Ch 1–9, 13**) and lecture
- Hint: think about solutions to the exercises in the book
- Questions: more than "how to call it in OpenGL":
  - ⋆ Basic concepts
  - ⋆ "How do things work?"
  - ⋆ Applications of techniques
- Check old exam on the WWW page!

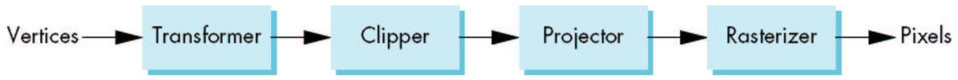
Course Summary / Exam Topics

1. Graphics Programming (OpenGL)
2. Input and Interaction
3. Geometric Objects and Transformations
4. Viewing (3D and perspectives)
5. Shading (light and matter)
6. Object Hierarchies
7. Discrete Techniques (texture)
8. Implementation of a Renderer

Synthetic Camera Model

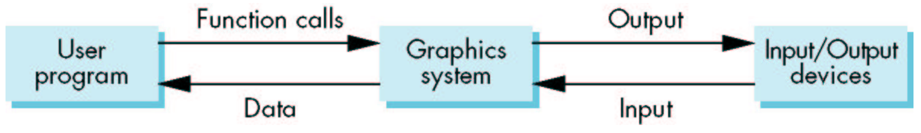


Pipeline Architecture

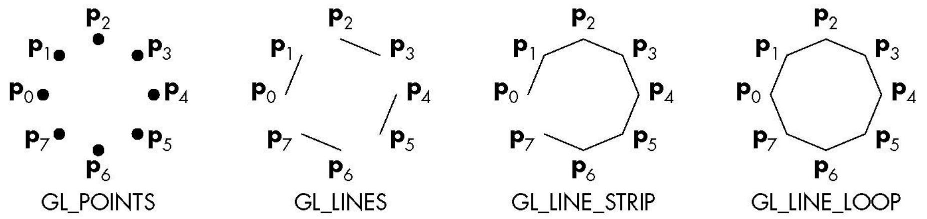


Graphics Functions

1. primitive functions (objects: “what”)
2. attribute functions “how”
3. viewing functions (camera)
4. transformation functions (rotation . . .)
5. input functions
6. control functions

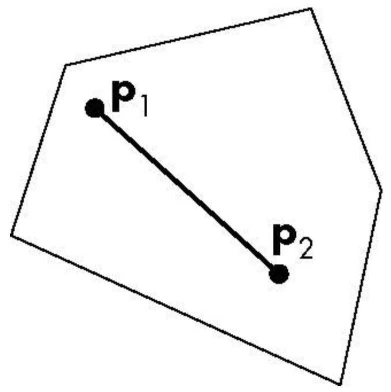


Primitive Elements



```
glBegin( ... );
  glVertex*( ... );
  .
  .
glEnd();
```

Convexity



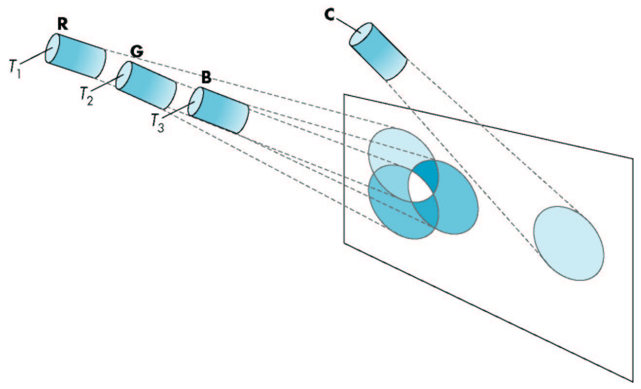
"All points on the line segment between any 2 points inside the polygon are inside the polygon."

Control and the Window System

```
#include <GL/glut.h>
int main(int argc, char** argv){
  glutInit(&argc,argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(0,0);
  glutCreateWindow("Sierpinski Gasket");
  glutDisplayFunc(display); /* register display func. */

  myinit();       /* application-specific inits */
  glutMainLoop(); /* enter event loop */
  return 0;
}
```

Additive Color Matching

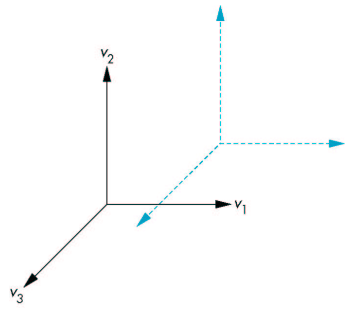


$$C = T_1 \cdot R + T_2 \cdot G + T_3 \cdot B$$

Double Buffering

- screen image is refreshed 50-85 times per second
- drawing into the frame buffer is not an atomic action
  - ⋆ (and takes longer than 1/50 sec)
- the flickering we see is from partially drawn images
- solution: double buffering
  - ⋆ front buffer is used for display
  - ⋆ back buffer is used for drawing

Frame: Basis Vectors + Reference Point



Vector: $v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$
Point: $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$

Homogeneous Coordinates

$$P = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + P_0$$

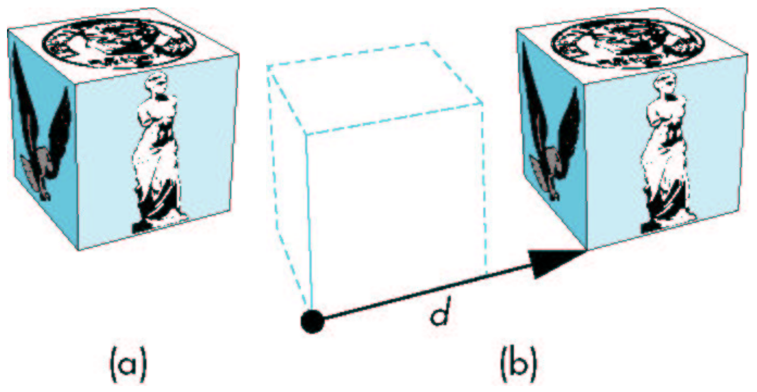
Define point-scalar "multiplication":

$$0 \cdot P = 0$$
$$1 \cdot P = P$$

$$P = [\alpha_1\ \alpha_2\ \alpha_3\ 1] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \qquad p = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{bmatrix}$$

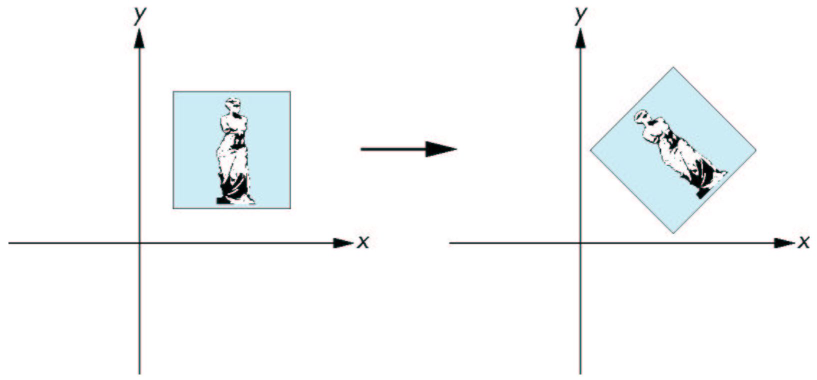
$$w = [\delta_1\ \delta_2\ \delta_3\ 0] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \qquad a = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ 0 \end{bmatrix}$$

Translation



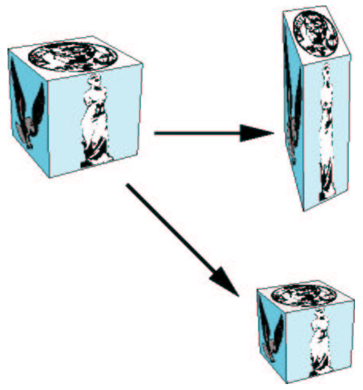
$$P' = P + d$$

Rotation



rotation (in 2D) about a fixed point

Scaling



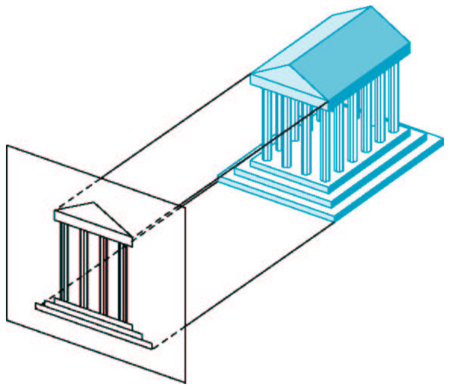
Rotation around axes

$$R_z = R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x = R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

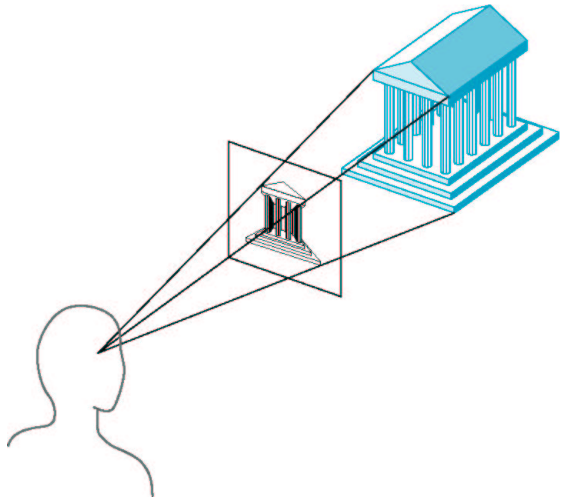
$$R_y = R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Orthographic Projection

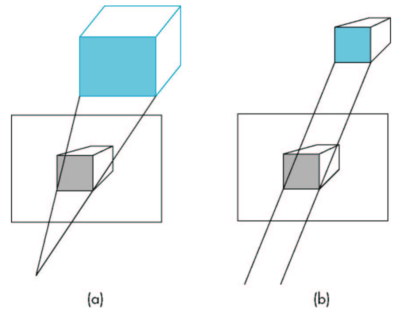


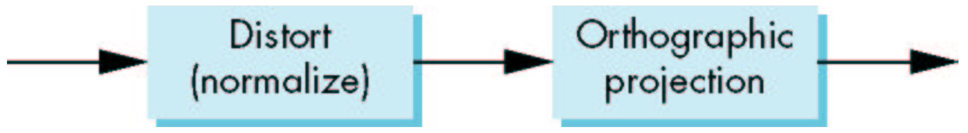
Projectors are perpendicular to the projection plane.

Perspective Viewing

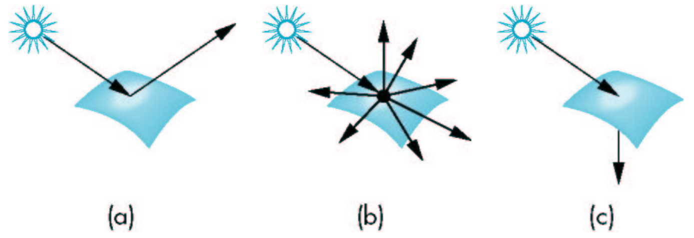


Predistortion of Objects



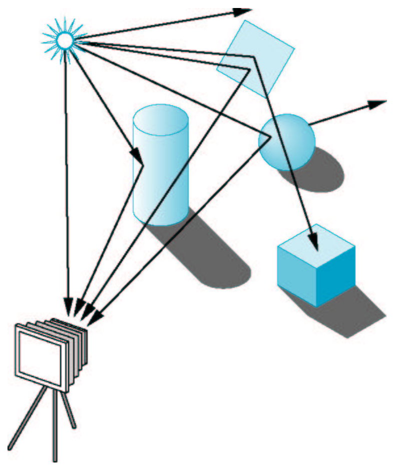


Light-Material Interactions

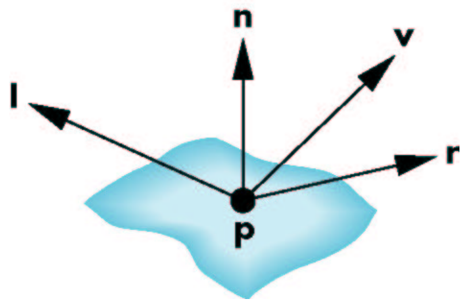


specular diffuse translucent

Global Rendering (Ray Tracing)

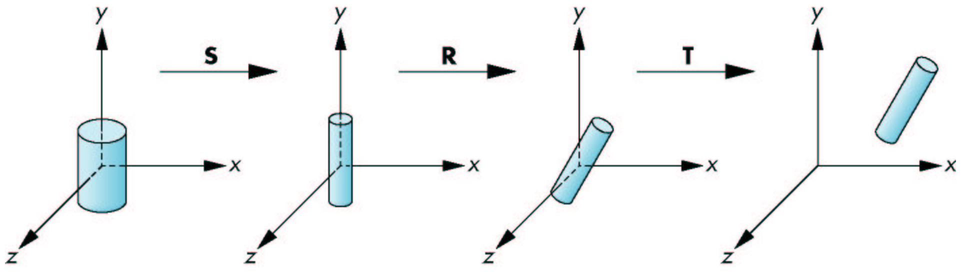


The Phong Reflection Model



n normal vector at p
v vector to viewer (COP)
l vector to light source
r direction of reflected light

Instance Transformation



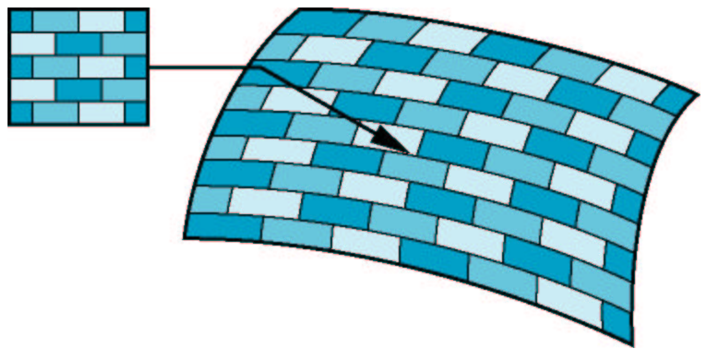
$M = TRS$

converts normalized object into customized instance

Traversing an Object Tree

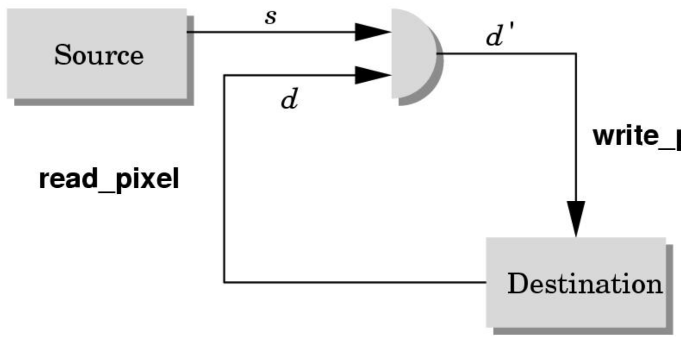
```
void traverse(treenode *root){
    if (root == NULL) return;
    glPushMatrix();
    glMultMatrix(root->m);
    root->f();
    if (root->child != NULL) traverse(root->child);
    glPopMatrix();
    if (root->sibling != NULL) traverse(root->sibling);
}
```

Texture Mapping



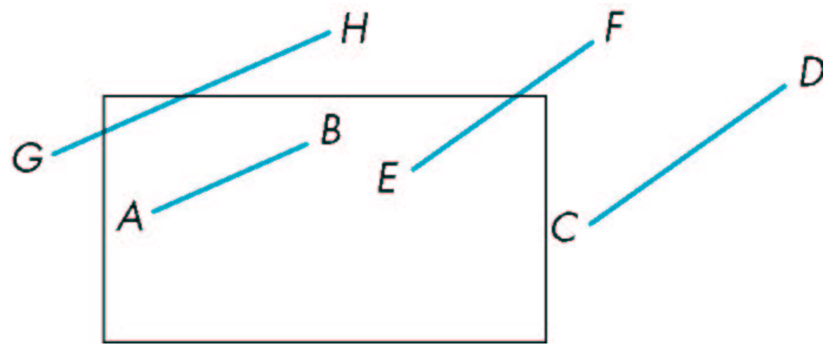
Pattern $\longrightarrow$ Surface

Buffer writing modes



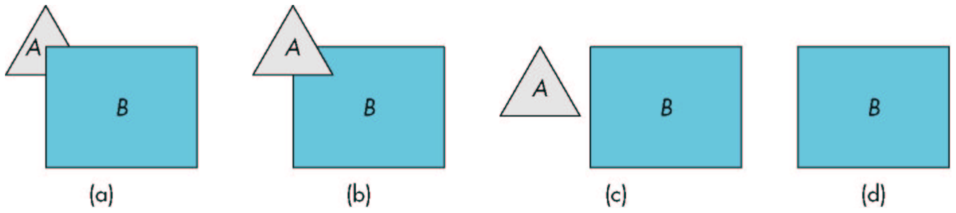
$d \leftarrow s$ $d \leftarrow s + d$

Line-Segment Clipping (start with 2D)



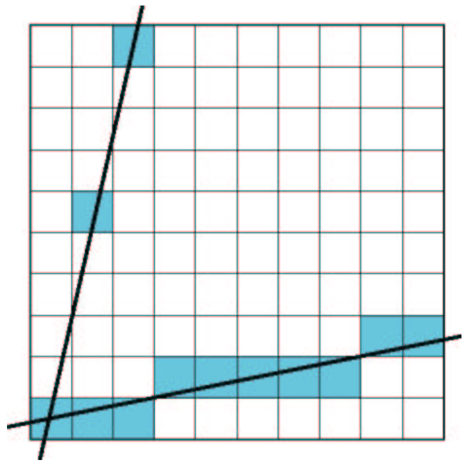
Accept or reject line segments
Shorten line segments.

Hidden-Surface Removal

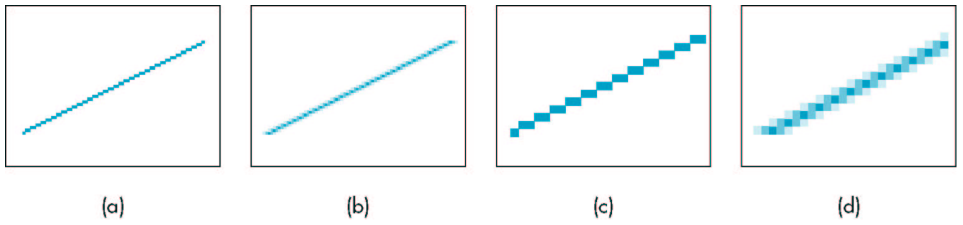


Object-space approach:
with k objects, compare each with $k-1$ objects:
$O(k^2)$ checks

Scan Conversion



Antialiasing



Finally . . .

- Thank you very much for being here!
- I wish you success with the programming projects and with the exam!