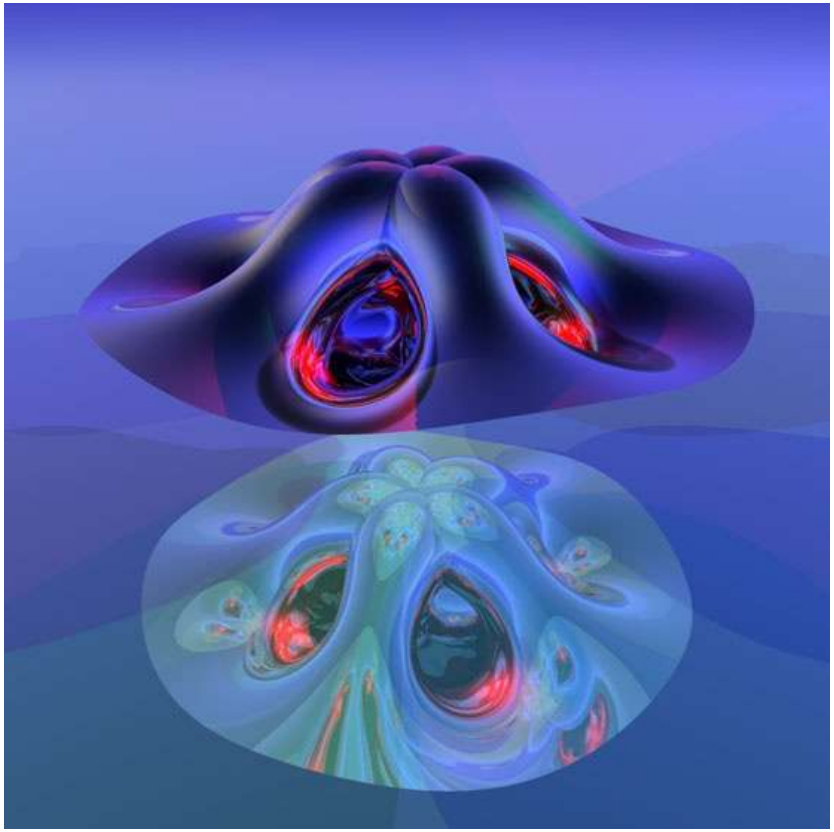
Modeling II + Rendering leftovers

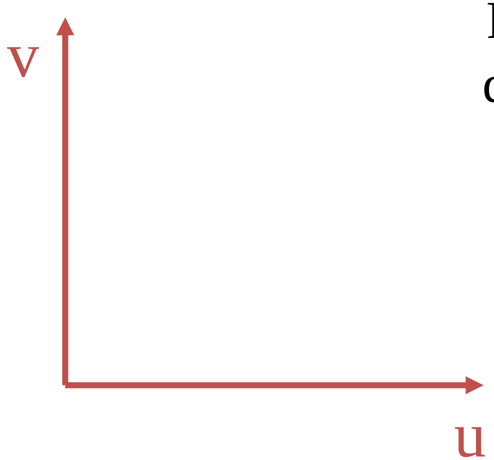
CG10a
Lior Shapira
Lecture 10



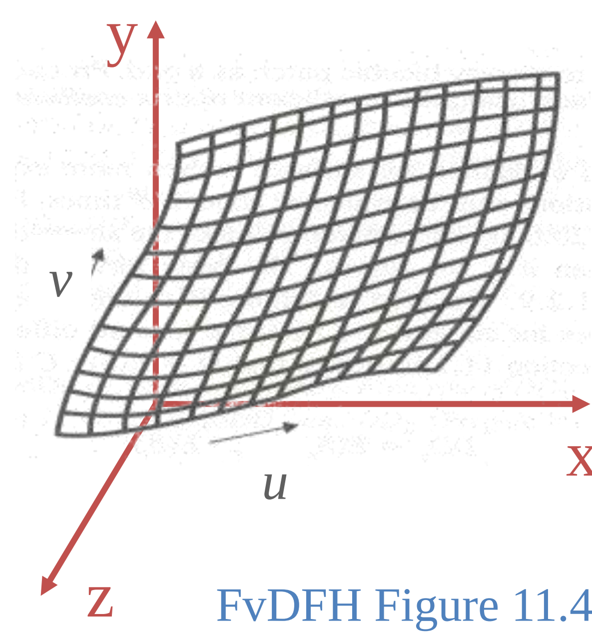
Some slide from Thomas Funkhouser, Princeton University

Parametric Surfaces

- Boundary defined by parametric functions:
  - $x = f_x(u,v)$
  - $y = f_y(u,v)$
  - $z = f_z(u,v)$



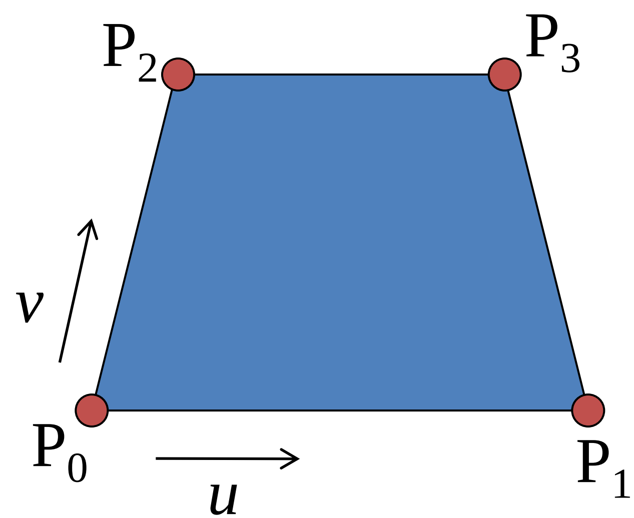
Parametric functions define mapping from (u,v) to (x,y,z):



FvDFH Figure 11.42

Parametric Surfaces

- Boundary defined by parametric functions:
  - $x = f_x(u,v)$
  - $y = f_y(u,v)$
  - $z = f_z(u,v)$



- Example: quadrilateral

$$f_x(u,v) = (1-v)((1-u)x_0 + ux_1) + v((1-u)x_2 + ux_3)$$

$$f_y(u,v) = (1-v)((1-u)y_0 + uy_1) + v((1-u)y_2 + uy_3)$$

$$f_z(u,v) = (1-v)((1-u)z_0 + uz_1) + v((1-u)z_2 + uz_3)$$

Parametric Surfaces

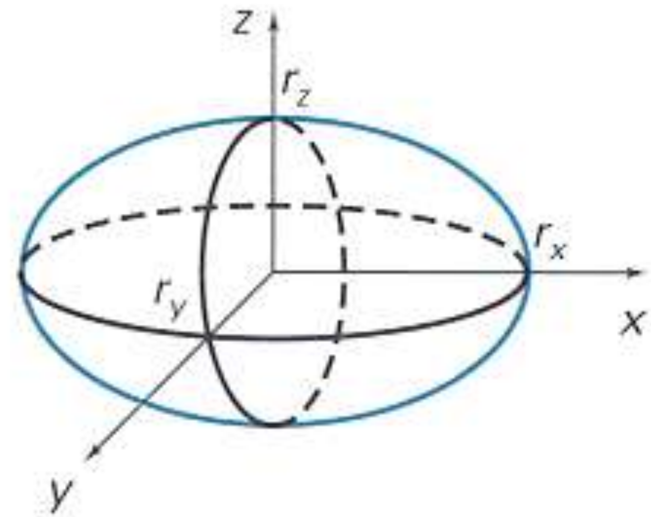
- Boundary defined by parametric functions:
  - $x = f_x(u,v)$
  - $y = f_y(u,v)$
  - $z = f_z(u,v)$

- Example: ellipsoid

$$f_x(u,v) = r_x \cos v \cos u$$

$$f_y(u,v) = r_y \cos v \sin u$$

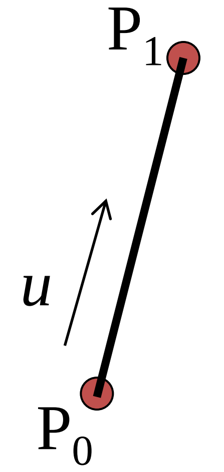
$$f_z(u,v) = r_z \sin v$$



H&B Figure 10.10

Parametric Curves

- Boundary defined by parametric functions:
  - $x = f_x(u)$
  - $y = f_y(u)$
- Example: line segment



$$f_x(u) = (1-u)x_0 + ux_1$$
$$f_y(u) = (1-u)y_0 + uy_1$$

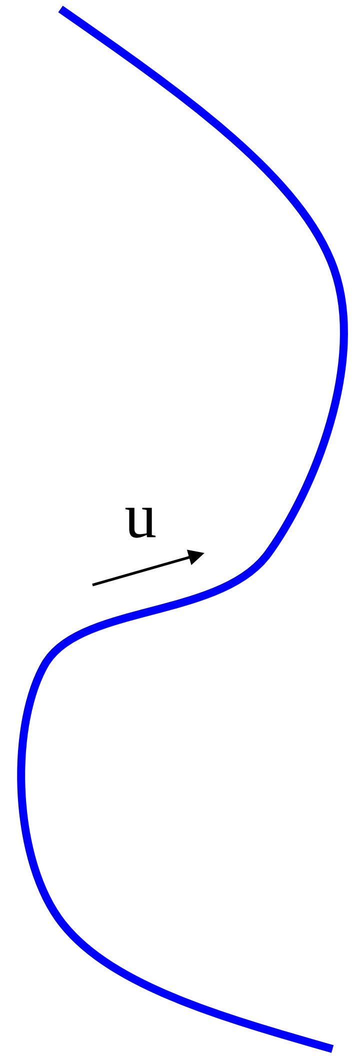
H&B Figure 10.10

Parametric curves

How can we define arbitrary curves?

$x = f_x(u)$

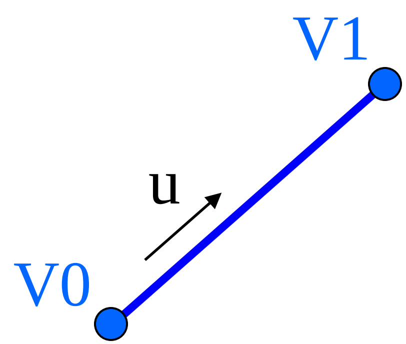
$y = f_y(u)$



Parametric curves

How can we define arbitrary curves?

$$x = f_x(u)$$
$$y = f_y(u)$$



Use functions that "blend" control points

$$x = f_x(u) = V0_x * (1 - u) + V1_x * u$$
$$y = f_y(u) = V0_y * (1 - u) + V1_y * u$$

Parametric curves

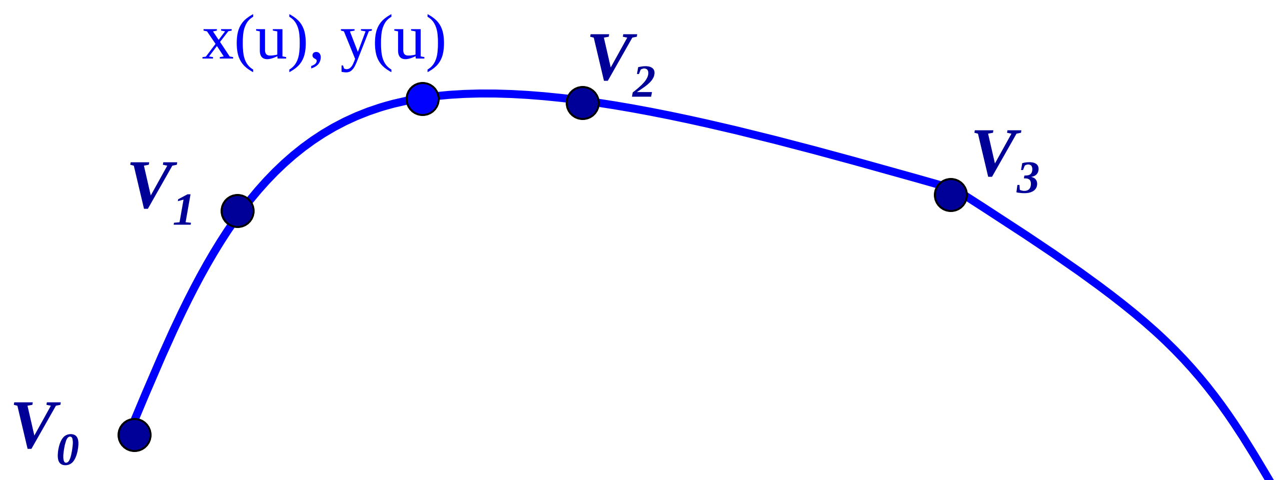
More generally:

$$x(u) = \sum_{i=0}^{n} B_i(u) * Vi_x$$

$$y(u) = \sum_{i=0}^{n} B_i(u) * Vi_y$$

Weights

Control Points



Parametric curves

What B(u) functions should we use?

$$x(u) = \sum_{i=0}^{n} B_i(u) * Vi_x$$

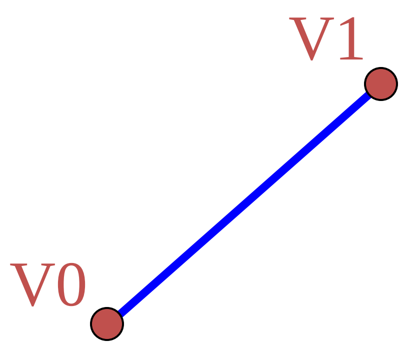
$$y(u) = \sum_{i=0}^{n} B_i(u) * Vi_y$$

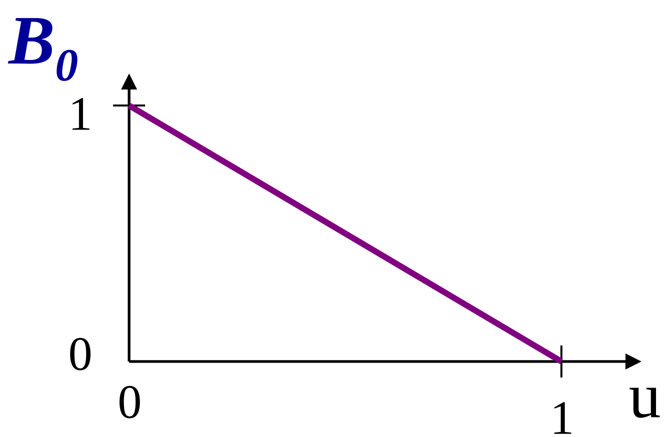
Parametric curves

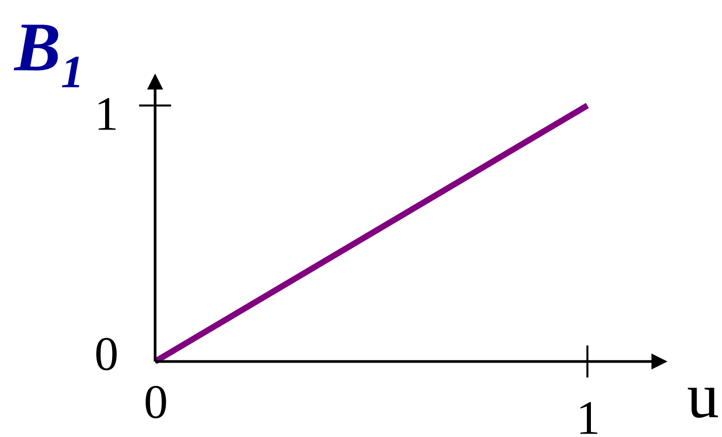
What B(u) functions should we use?

$$x(u) = \sum_{i=0}^{n} B_i(u) * Vi_x$$

$$y(u) = \sum_{i=0}^{n} B_i(u) * Vi_y$$





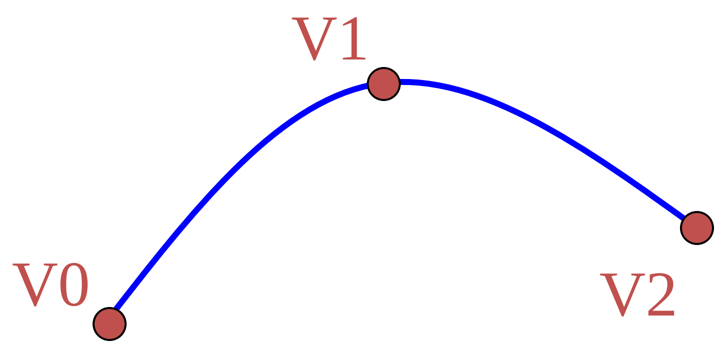


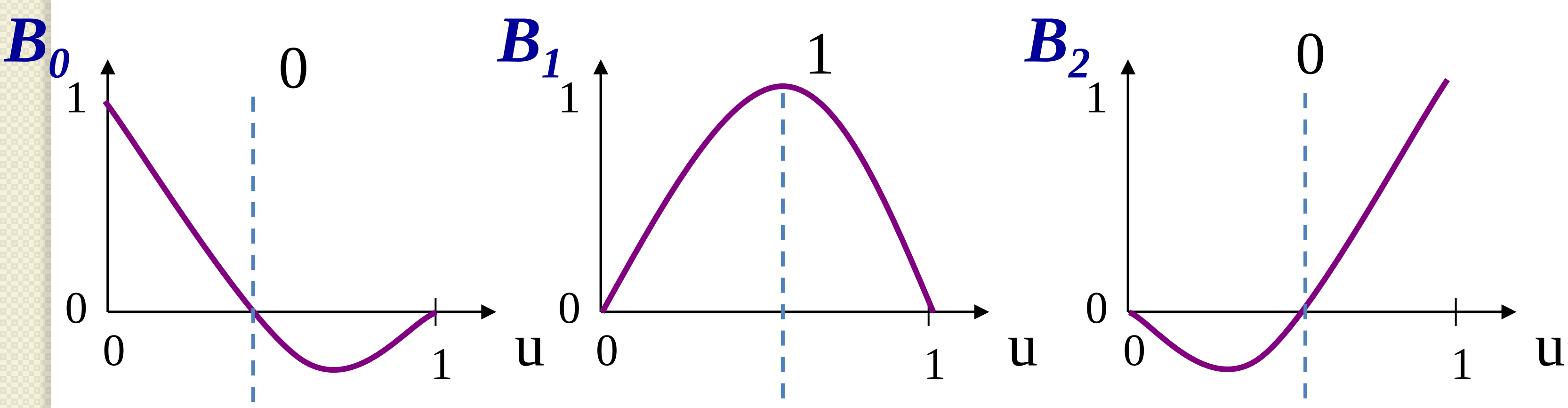
Parametric curves

What B(u) functions should we use?

$$x(u) = \sum_{i=0}^{n} B_i(u) * Vi_x$$

$$y(u) = \sum_{i=0}^{n} B_i(u) * Vi_y$$

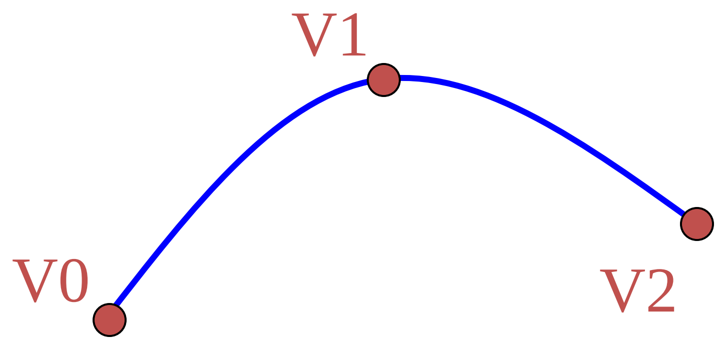




Parametric polynomial curves

- Polynomial blending functions:

$$B_i(u) = \sum_{j=0}^{m} a_j u^j$$

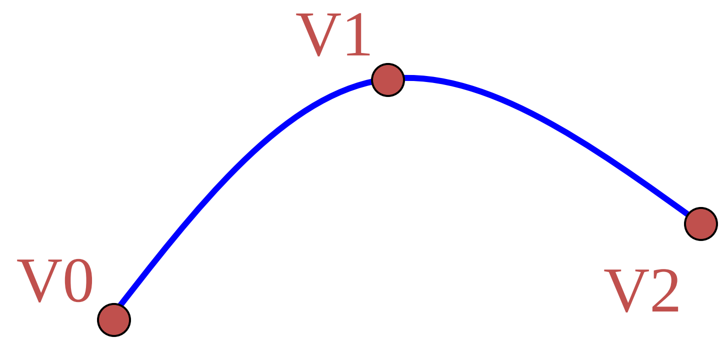


- Advantages of polynomials
  - Easy to compute
  - Infinitely continuous
  - Easy to derive curve properties

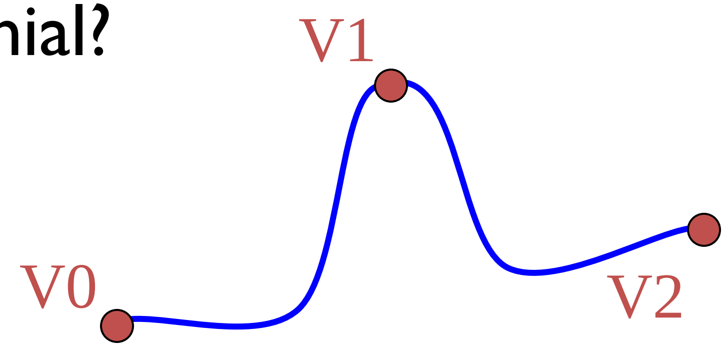
Parametric polynomial curves

- Polynomial blending functions:

$$B_i(u) = \sum_{j=0}^{m} a_j u^j$$

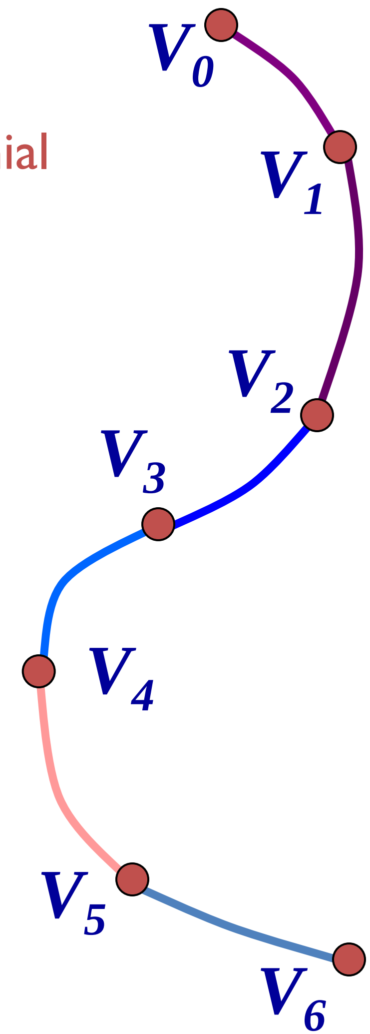


- What degree polynomial?
  - Easy to compute
  - Easy to control
  - Expressive



Piecewise parametric polynomial curves

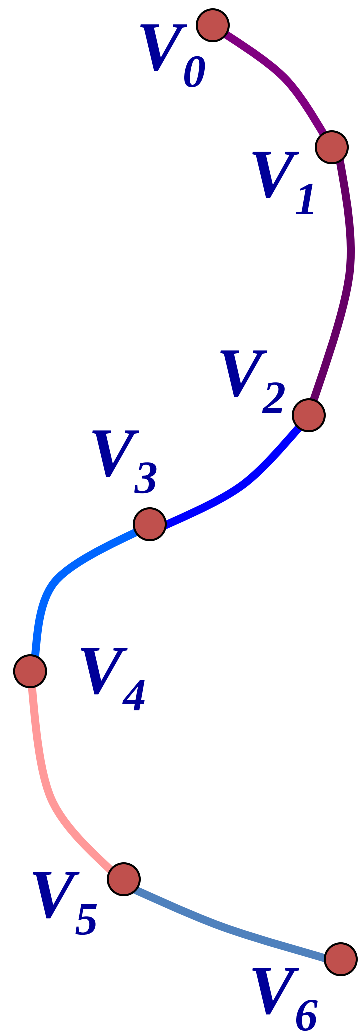
- Splines:
  - Split curve into segments
  - Each segment defined by **low-order** polynomial blending subset of control vertices
- Motivation:
  - Provides control & efficiency
  - Same blending function for every segment
  - Prove properties from blending functions
- Challenges
  - How to choose blending functions?
  - How to determine properties?



Cubic Splines

- Some properties we might like to have:
  - Local control
  - Interpolation
  - Continuity
  - Convex hull

$$B_i(u) = \sum_{j=0}^{m} a_j u^j$$

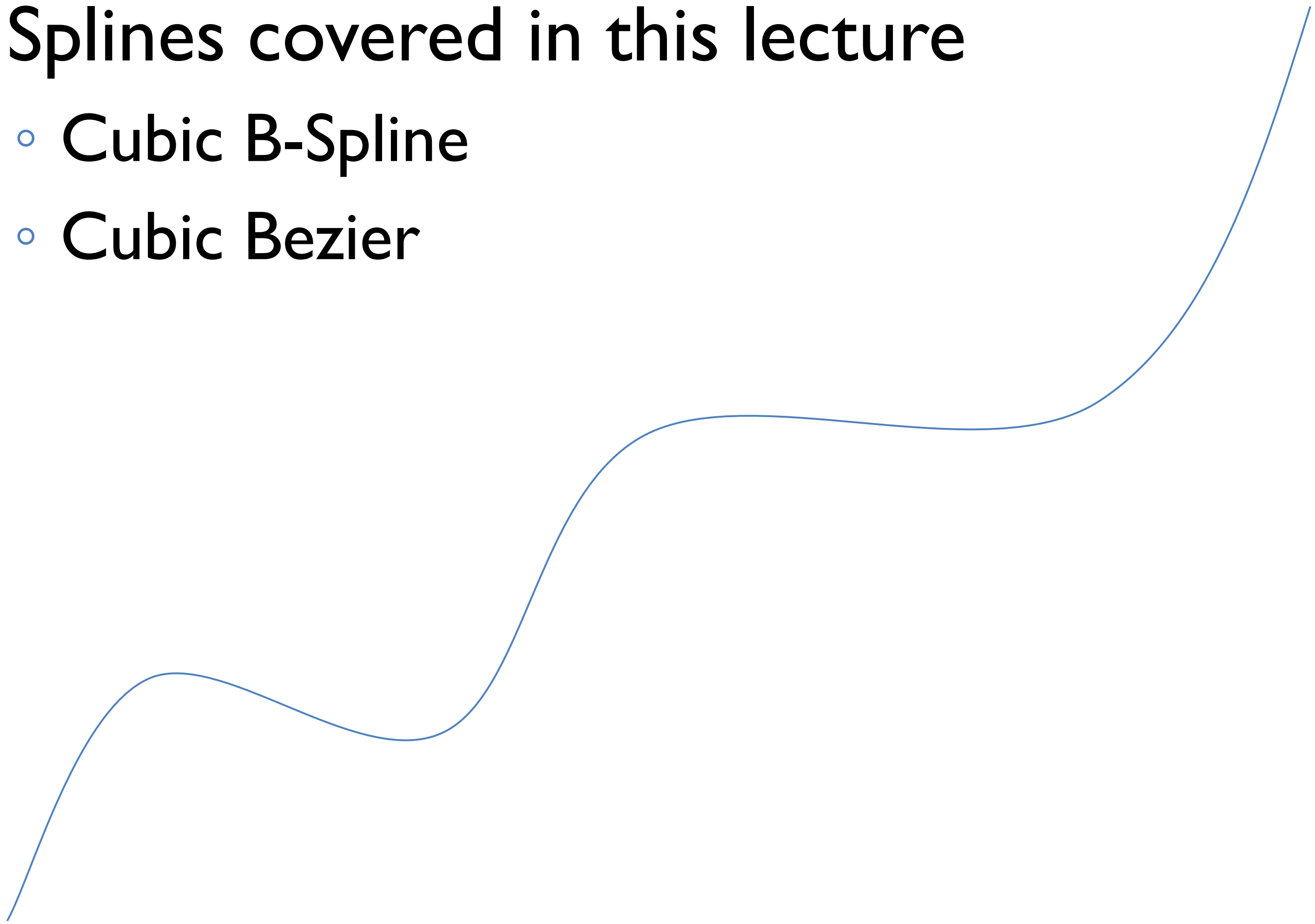


Blending functions determine properties

Properties determine blending functions

Cubic Splines

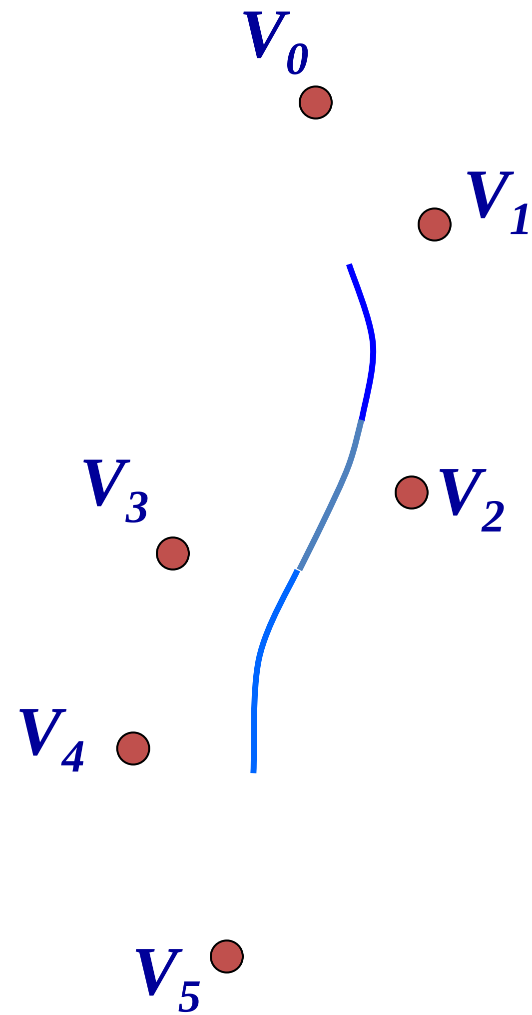
- Splines covered in this lecture
  - Cubic B-Spline
  - Cubic Bezier



** There are many others*

Cubic B-Splines

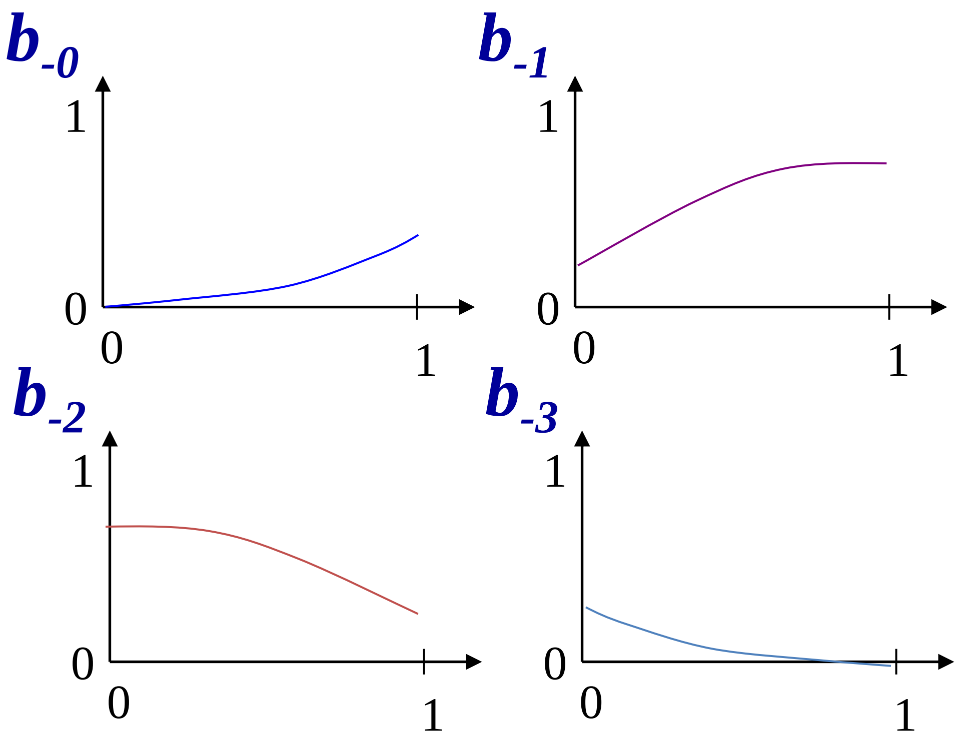
- Properties:
  1. Local control
  2. C^2 continuity
  3. Approximating
  4. Convex hull

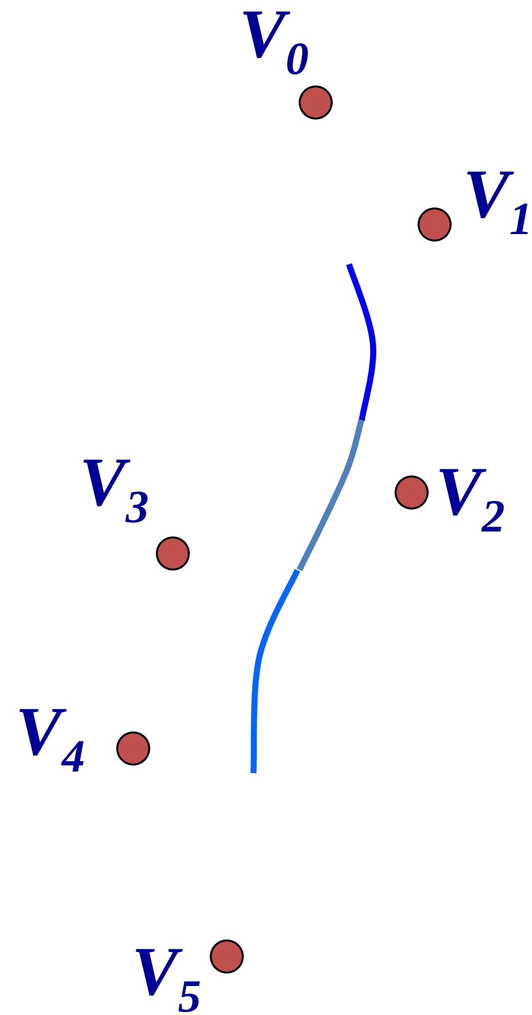


Cubic B-Spline Blending Functions

Blending functions:

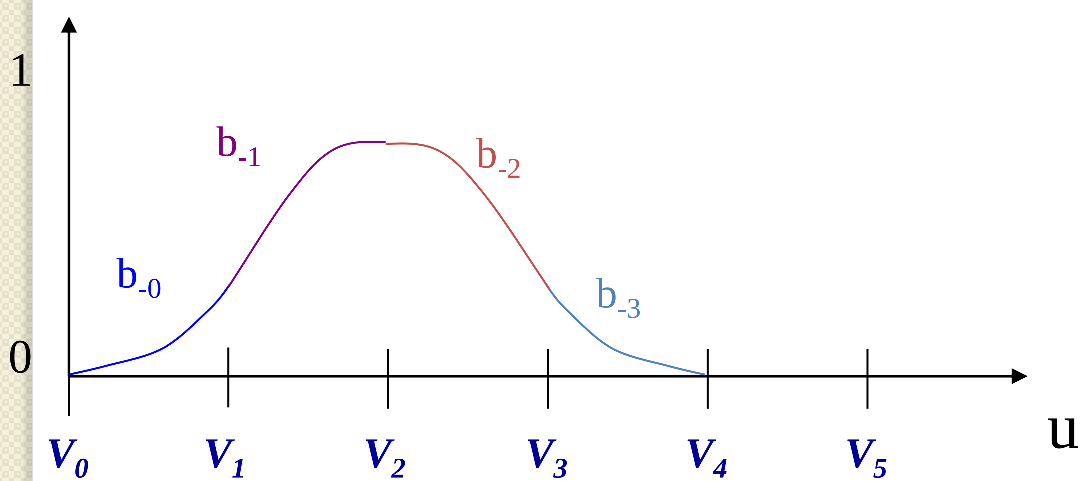
$$B_i(u) = \sum_{j=0}^{m} a_j u^j$$

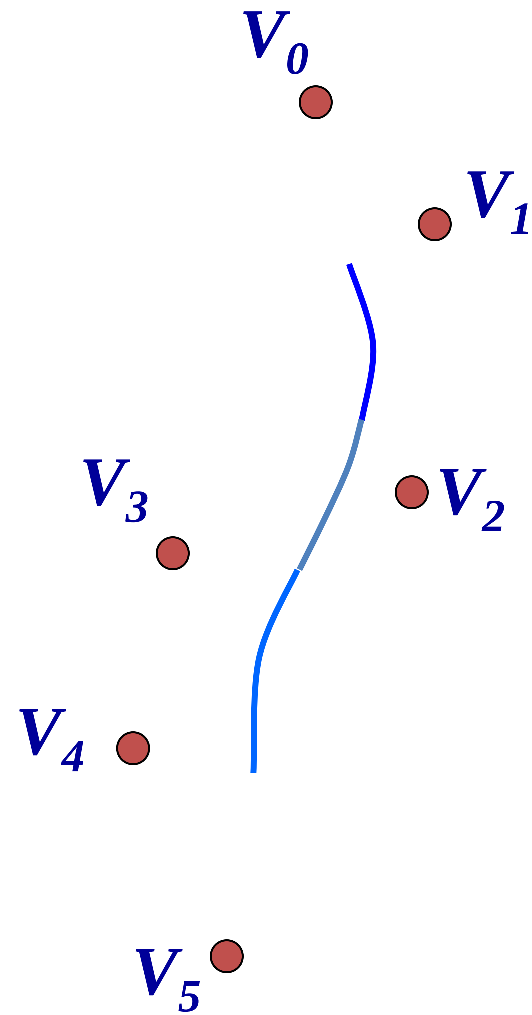




Cubic B-Spline Blending Functions

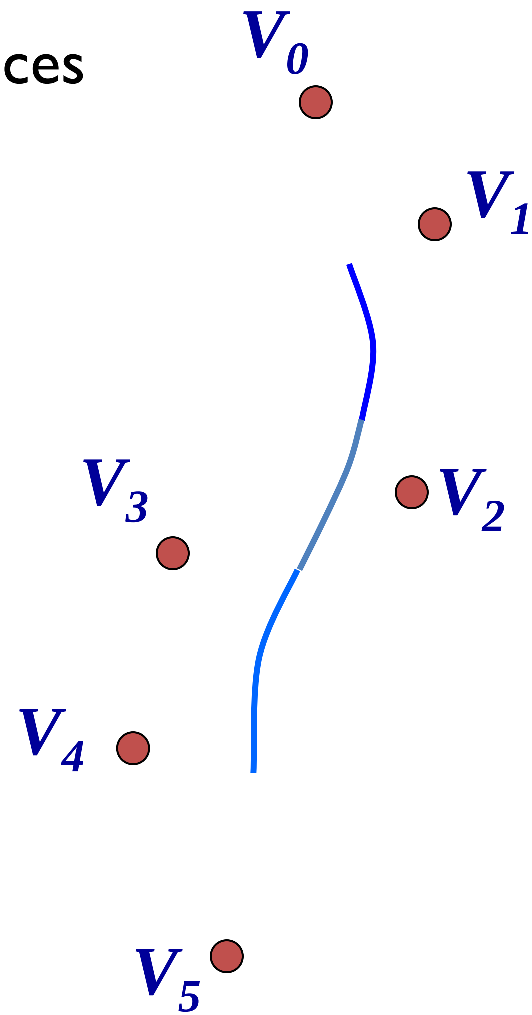
- How to derive blending functions?
  - Cubic polynomials
  - Local control
  - C^2 continuity
  - Convex hull





Cubic B-Spline Blending Functions

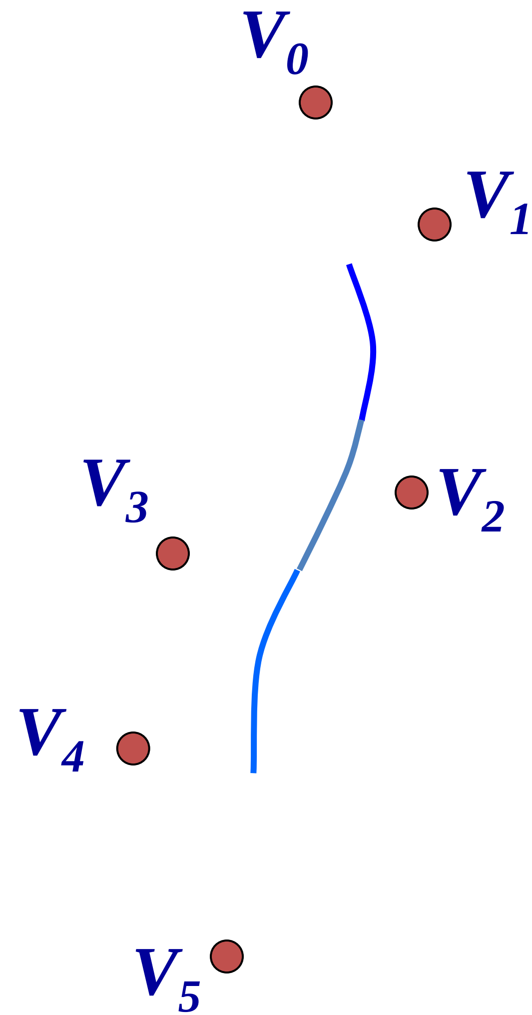
- Four cubic polynomials for four vertices
  - 16 variables (degrees of freedom)
  - Variables are a_i, b_i, c_i, d_i for four blending functions



$$
\boxed{
\begin{aligned}
b_{-0}(u) &= a_0u^3 + b_0u^2 + c_0u^1 + d_0 \\
b_{-1}(u) &= a_1u^3 + b_1u^2 + c_1u^1 + d_1 \\
b_{-2}(u) &= a_2u^3 + b_2u^2 + c_2u^1 + d_2 \\
b_{-3}(u) &= a_3u^3 + b_3u^2 + c_3u^1 + d_3
\end{aligned}
}
$$

Cubic B-Spline Blending Functions

- C2 continuity implies 15 constraints
  - Position of two curves same
  - Derivative of two curves same
  - Second derivatives same



Cubic B-Spline Blending Functions

Fifteen continuity constraints:

$$0 = b_{-0}(0)$$
$$b_{-0}(1) = b_{-1}(0)$$
$$b_{-1}(1) = b_{-2}(0)$$
$$b_{-2}(1) = b_{-3}(0)$$
$$b_{-3}(1) = 0$$

$$0 = b_{-0}'(0)$$
$$b_{-0}'(1) = b_{-1}'(0)$$
$$b_{-1}'(1) = b_{-2}'(0)$$
$$b_{-2}'(1) = b_{-3}'(0)$$
$$b_{-3}'(1) = 0$$

$$0 = b_{-0}''(0)$$
$$b_{-0}''(1) = b_{-1}''(0)$$
$$b_{-1}''(1) = b_{-2}''(0)$$
$$b_{-2}''(1) = b_{-3}''(0)$$
$$b_{-3}''(1) = 0$$

One more convenient constraint:

$$b_{-0}(0) + b_{-1}(0) + b_{-2}(0) + b_{-3}(0) = 1$$

Cubic B-Spline Blending Functions

- Solving the system of equations yields:

$$
\begin{aligned}
b_{-3}(u) &= -\tfrac{1}{6}u^3 + \tfrac{1}{2}u^2 - \tfrac{1}{2}u + \tfrac{1}{6} \\
b_{-2}(u) &= \tfrac{1}{2}u^3 - u^2 + \tfrac{2}{3} \\
b_{-1}(u) &= -\tfrac{1}{2}u^3 + \tfrac{1}{2}u^2 + \tfrac{1}{2}u + \tfrac{1}{6} \\
b_{-0}(u) &= \tfrac{1}{6}u^3
\end{aligned}
$$

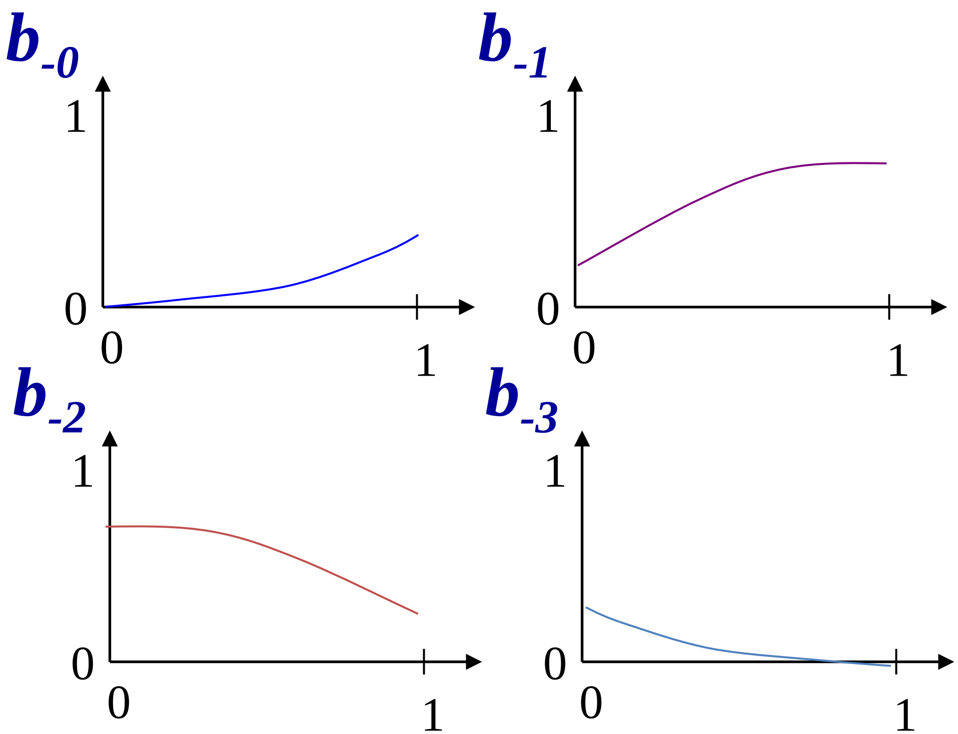
- In matrix form

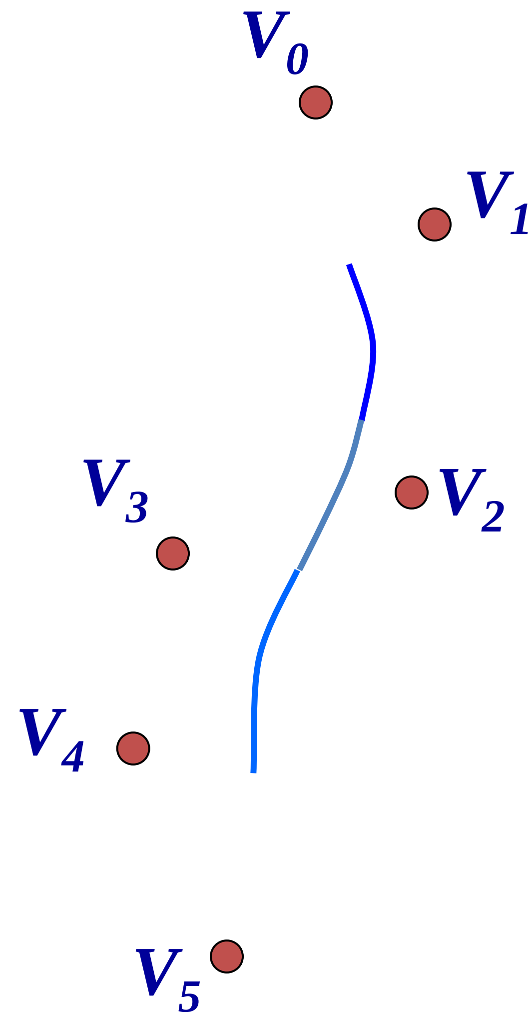
$$
Q(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{pmatrix}
$$

Cubic B-Spline Blending Functions

In plot form:

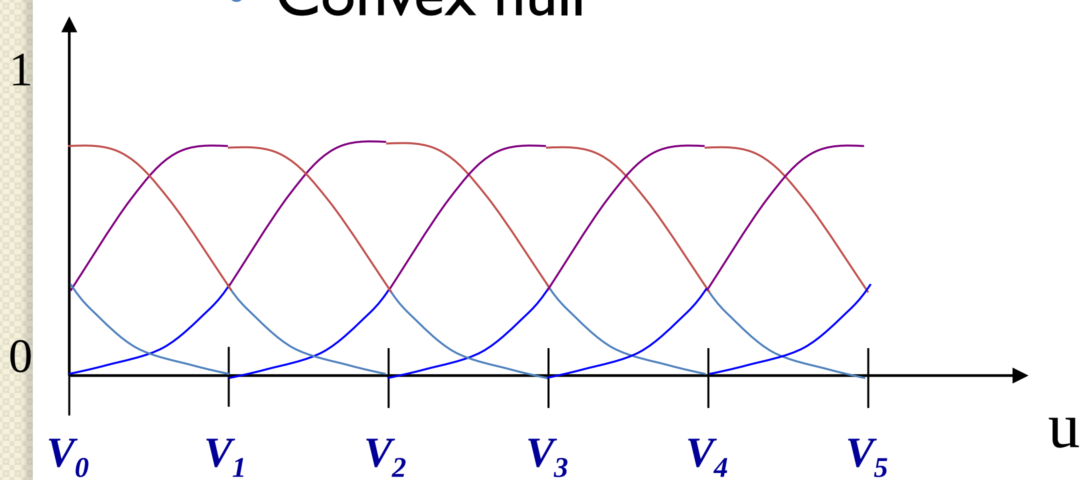
$$B_i(u) = \sum_{j=0}^{m} a_j u^j$$

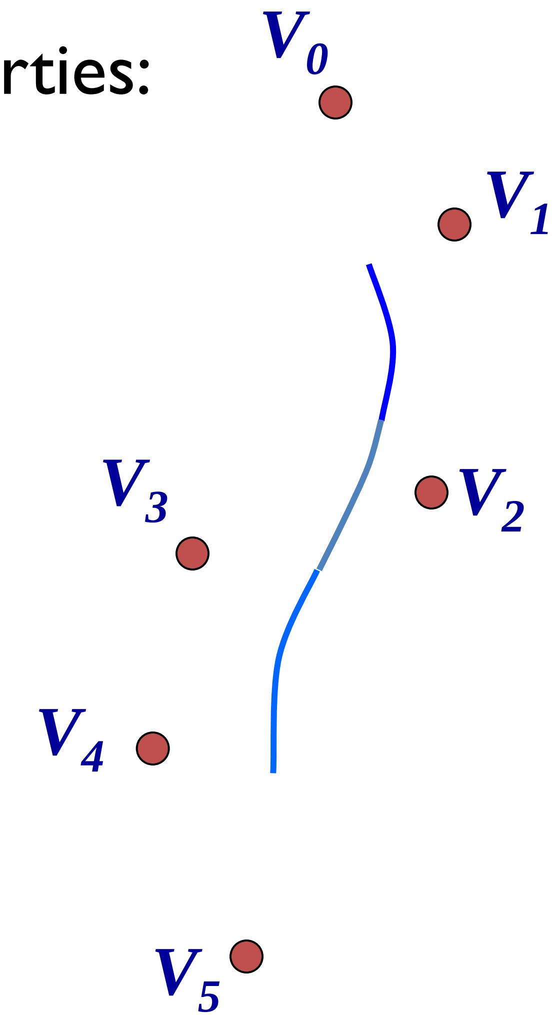




Cubic B-Spline Blending Functions

- Blending functions imply properties:
  - Local control
  - Approximating
  - C^2 continuity
  - Convex hull





Cubic Splines

- Splines covered in this lecture
  - Cubic B-Spline
  - Cubic Bezier

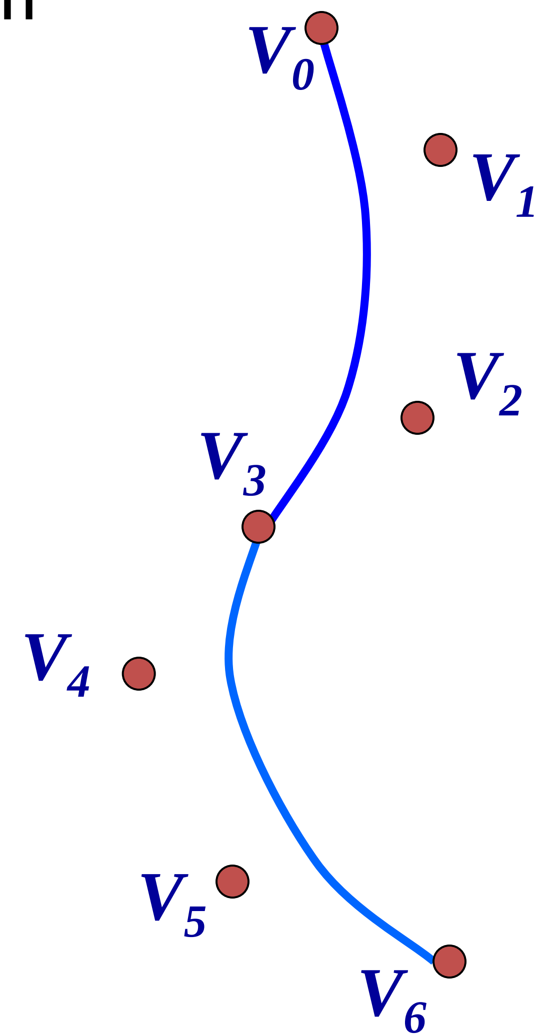
Properties determine blending functions

⇕

Blending functions determine properties

Cubic Bezier

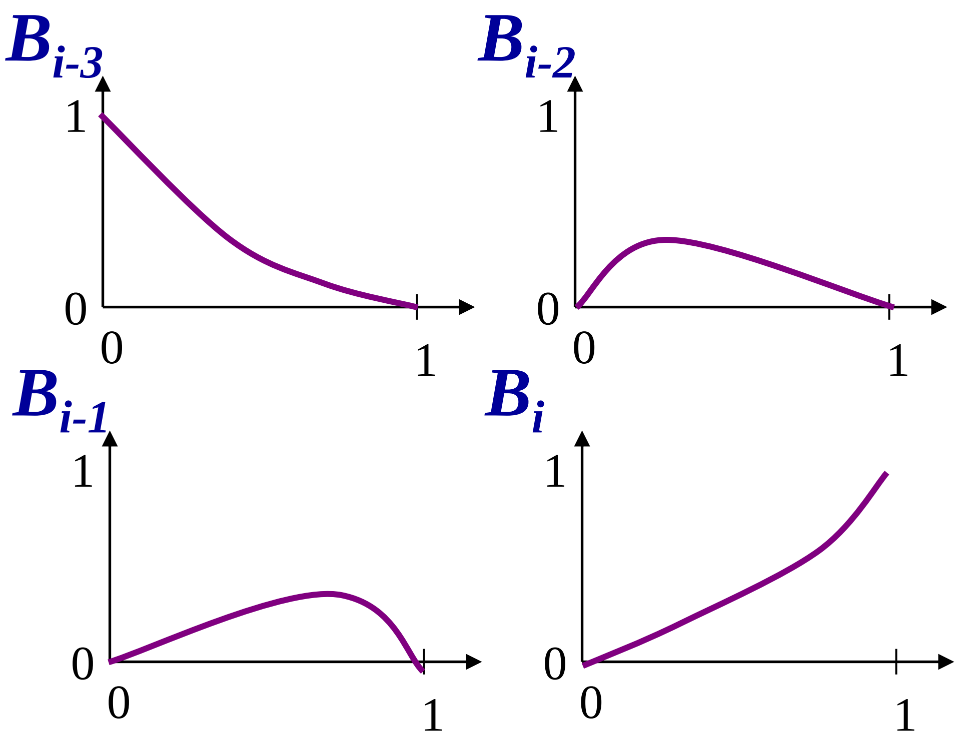
- Developed around 1960 by both
  - Bezier (Renault)
  - deCasteljau (Citroen)
- Properties:
  - Local control
  - C^1 continuity
  - Interpolating (every third)

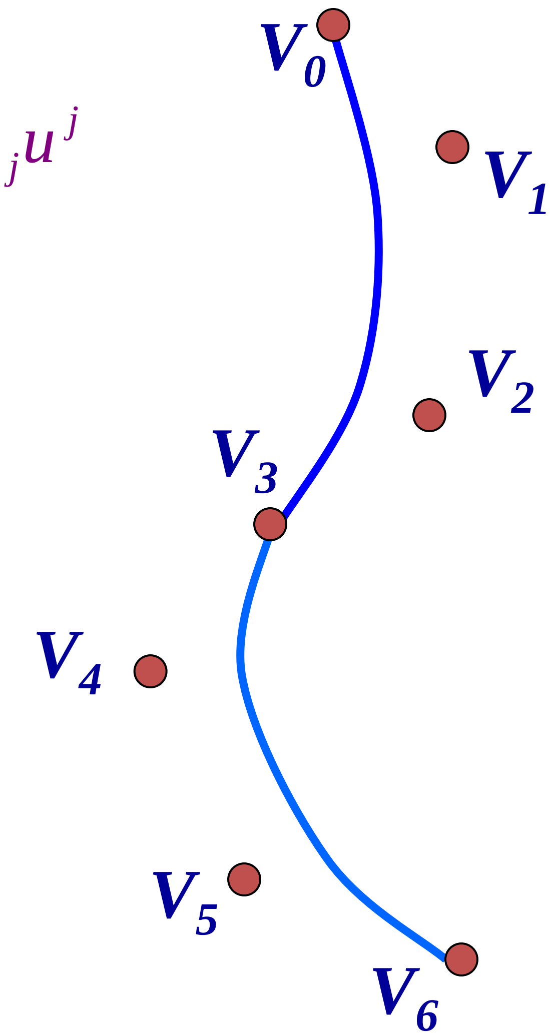


Cubic Bezier curves

Blending functions:

$$B_i(u) = \sum_{j=0}^{m} a_j u^j$$





Cubic Bezier Curves

Bézier curves in matrix form:

$$Q(u) = \sum_{i=0}^{n} V_i \binom{n}{i} u^i (1-u)^{n-i}$$

$$= (1-u)^3 V_0 + 3u(1-u)^2 V_1 + 3u^2(1-u) V_2 + u^3 V_3$$

$$= \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \underbrace{\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}}_{\mathbf{M}_{\mathbf{Bezier}}} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{pmatrix}$$

Basic properties of Bézier curves

- Endpoint interpolation:

$$Q(0) = V_0$$
$$Q(1) = V_n$$

- Convex hull:
  - Curve is contained within convex hull of control polygon

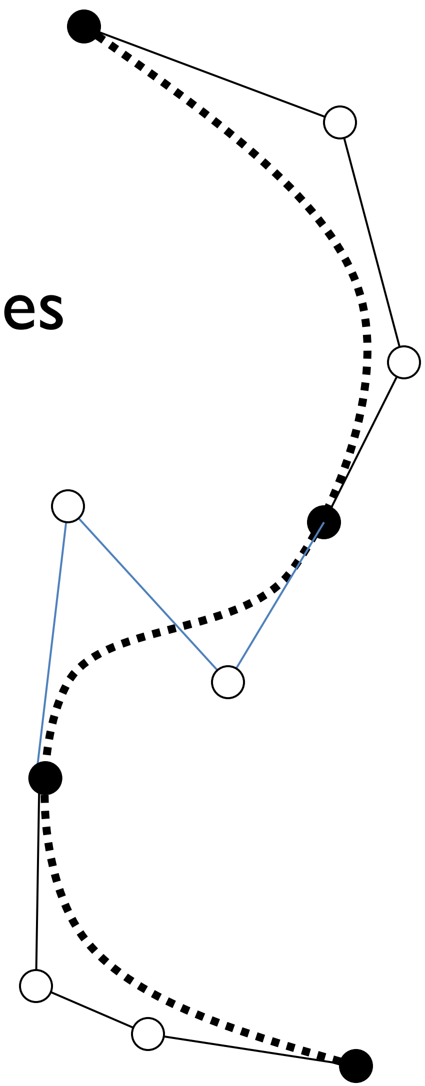
- Symmetry

$$Q(u) \text{ defined by } \{V_0, \ldots, V_n\} \equiv Q(1-u) \text{ defined by } \{V_n, \ldots, V_0\}$$

Bezier Splines

- For more complex curves, piece together Bézier curves

- Solve for "interior" control vertices
    - Positional (C^0) continuity
    - Derivative (C^1) continuity



Splines

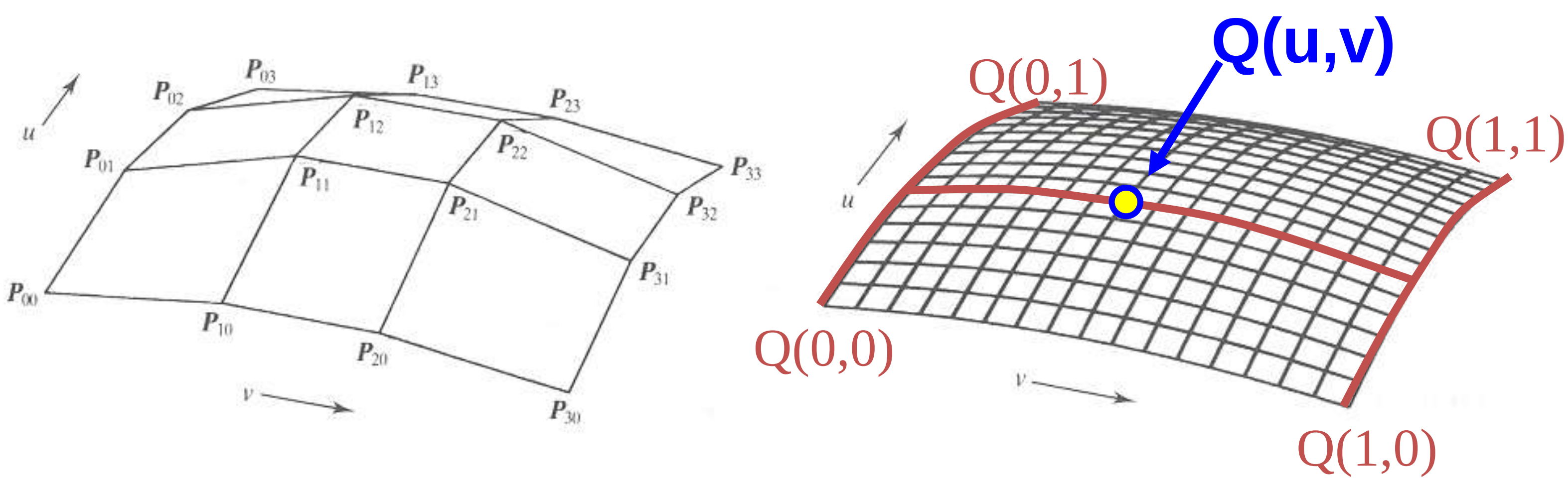
- Mathematical way to express curves
- Motivated by "loftsman's spline"
  - Long, narrow strip of wood/plastic
  - Used to fit curves through specified data points
  - Shaped by lead weights called "ducks"
  - Gives curves that are "smooth" or "fair"
- Have been used to design:
  - Automobiles
  - Ship hulls
  - Aircraft fuselage/wing

Summary

- Parametric Curves
  - Splines
  - Blending functions
    - Polynomial
      - B-Spline
      - Bezier

What's next?

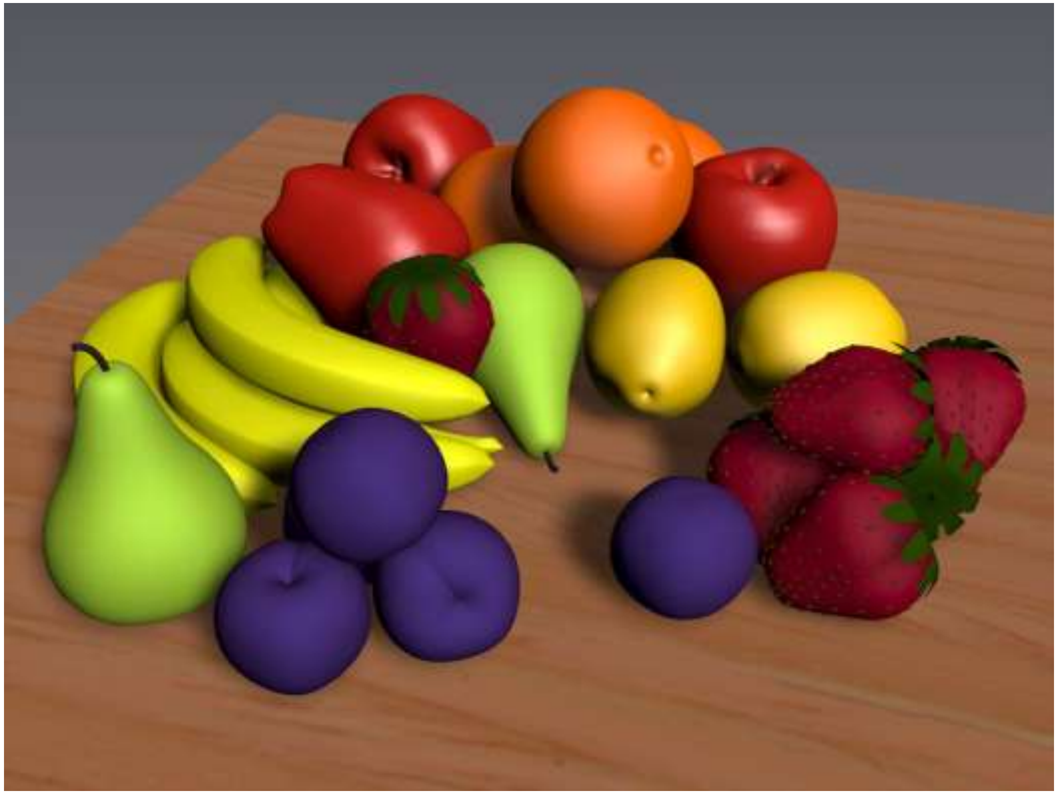
- Use curves to create parameterized surfaces



Watt Figure 6.21

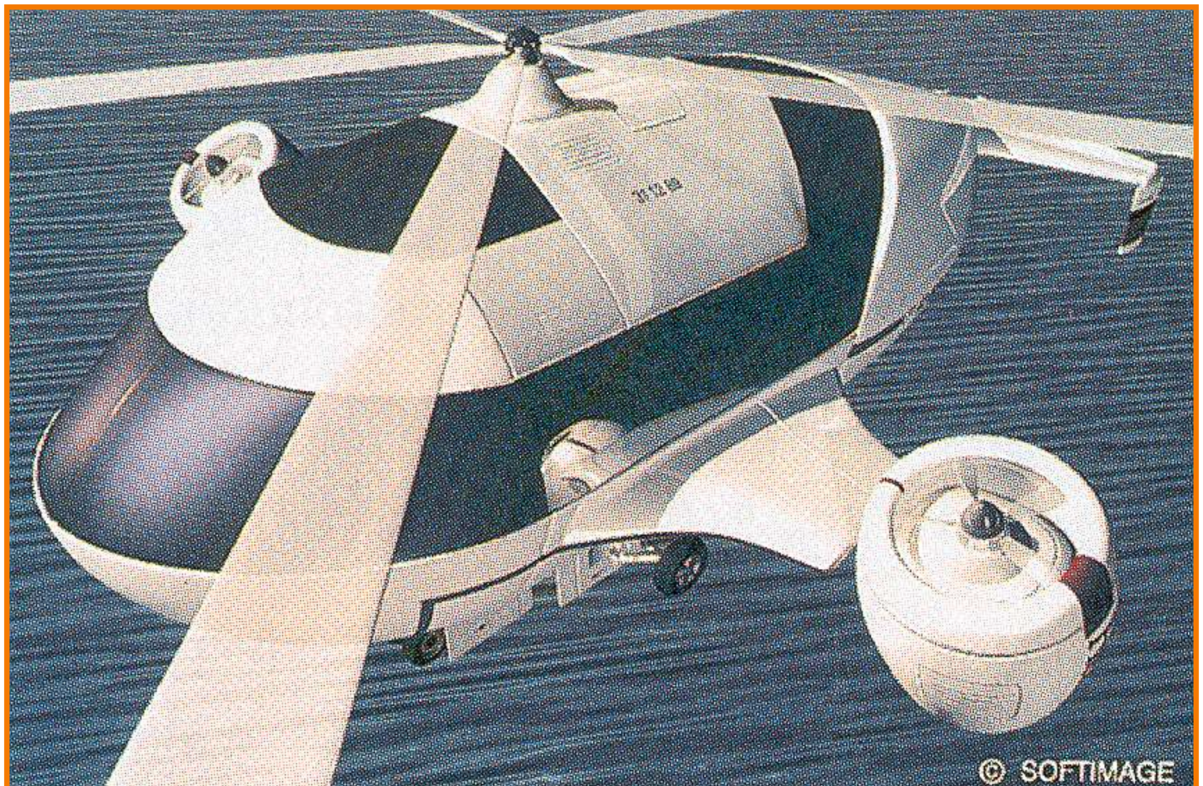
מה היום

- Curves
- **Surfaces**



Parametric Surfaces

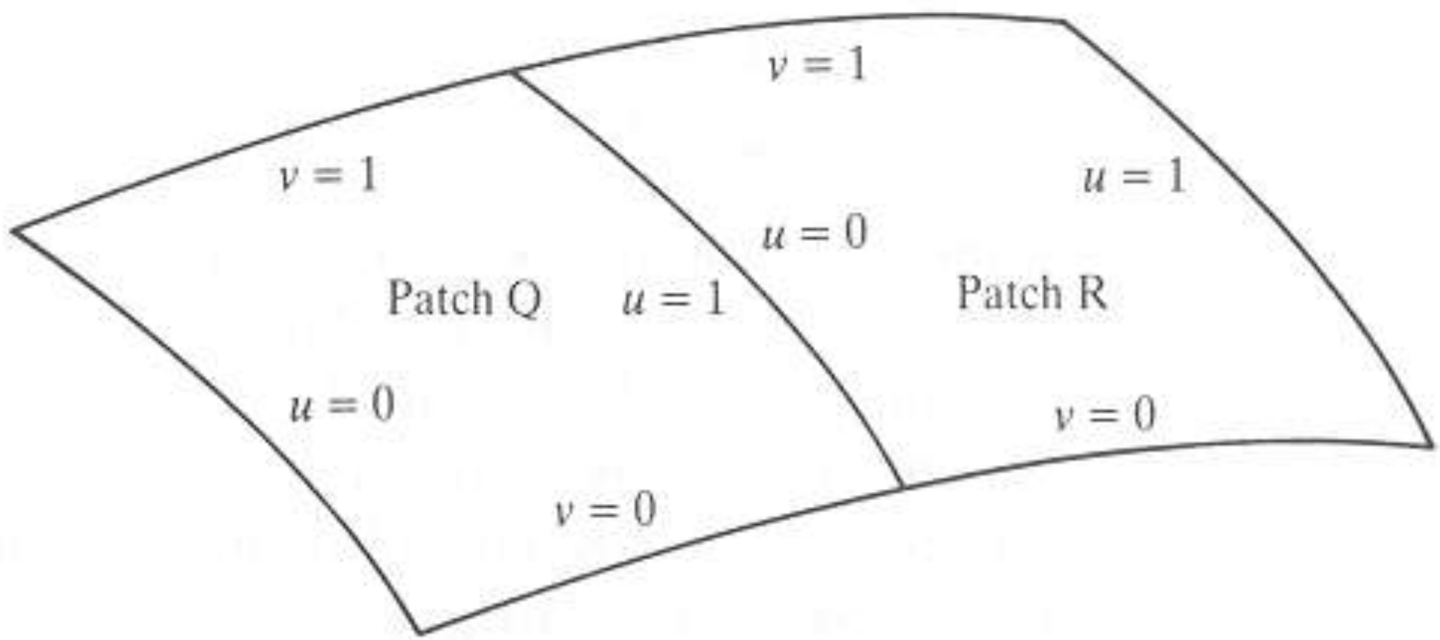
- How do we describe arbitrary smooth surfaces with parametric functions?



H&B Figure 10.46

Piecewise Polynomial Parametric Surfaces

- Surface is partitioned into parametric patches:

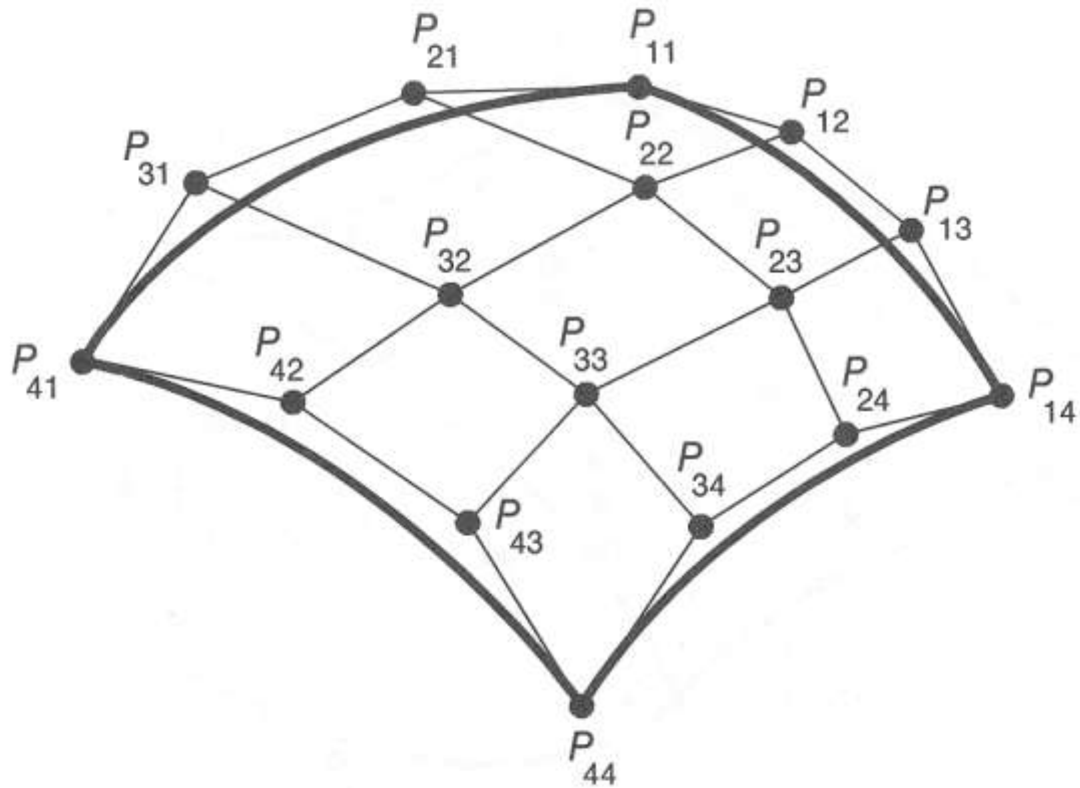


Same ideas as parametric splines!

Watt Figure 6.25

Parametric Patches

- Each patch is defined by blending control points



Same ideas as parametric curves!

FvDFH Figure 11.44

Parametric Bicubic Patches

Point Q(u,v) on any patch is defined by combining control points with polynomial blending functions:

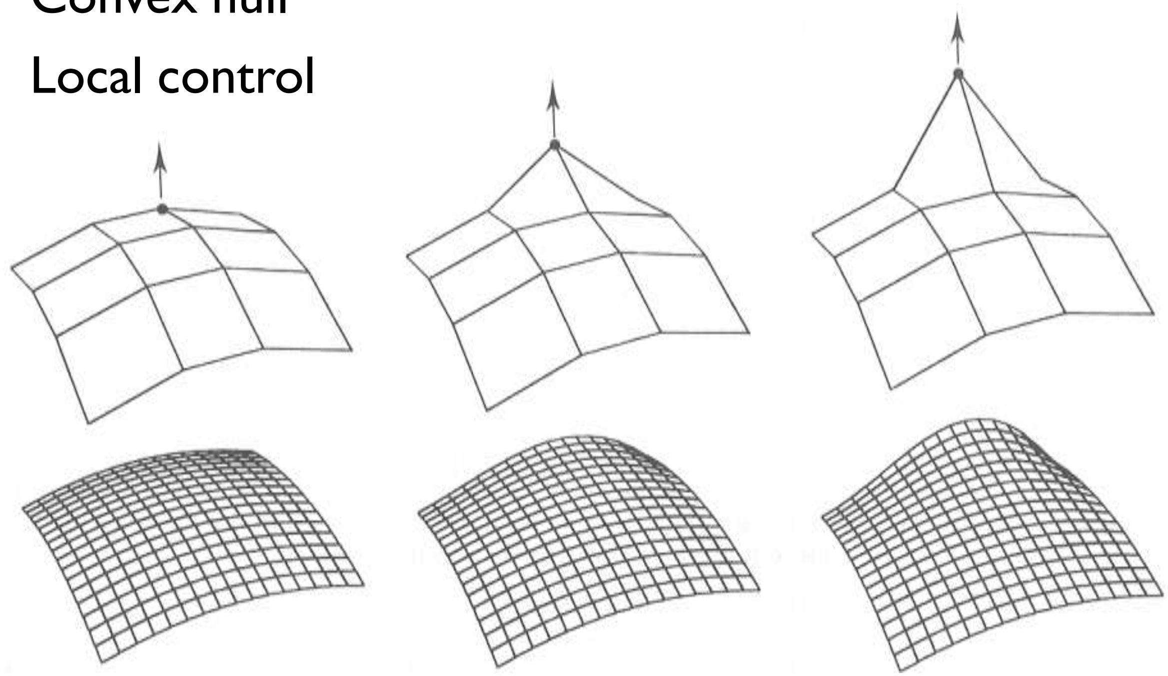
$$Q(u,v) = \mathbf{UM}\begin{bmatrix} P_{1,1} & P_{1,2} & P_{1,3} & P_{1,4} \\ P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} \\ P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} \\ P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} \end{bmatrix}\mathbf{M}^{\mathrm{T}}\mathbf{V}^{\mathrm{T}}$$

$$\mathbf{U} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \qquad \mathbf{V} = \begin{bmatrix} v^3 & v^2 & v & 1 \end{bmatrix}$$

Where M is a matrix describing the blending functions for a parametric cubic curve (e.g., Bezier, B-spline, etc.)

Bezier Patches

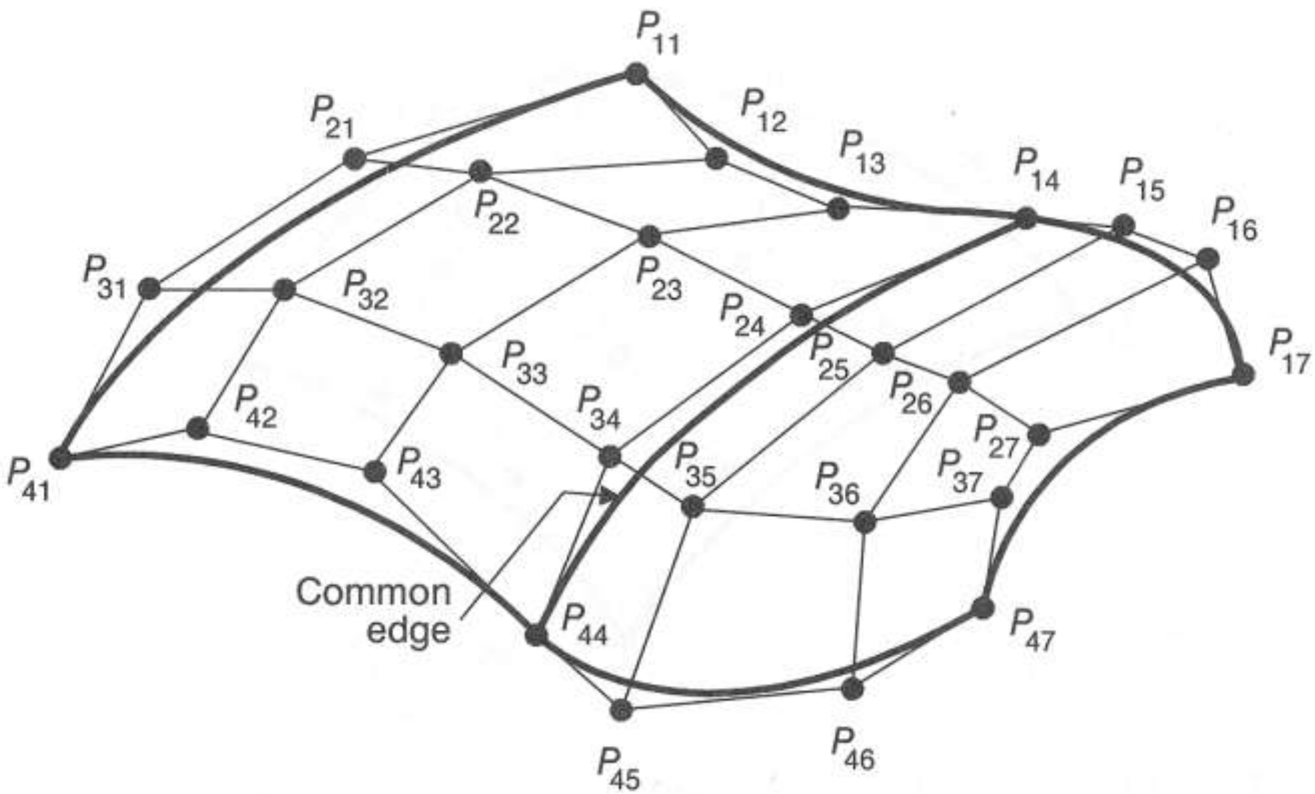
- Properties:
  - Interpolates four corner points
  - Convex hull
  - Local control



Watt Figure 6.22

Bezier Surfaces

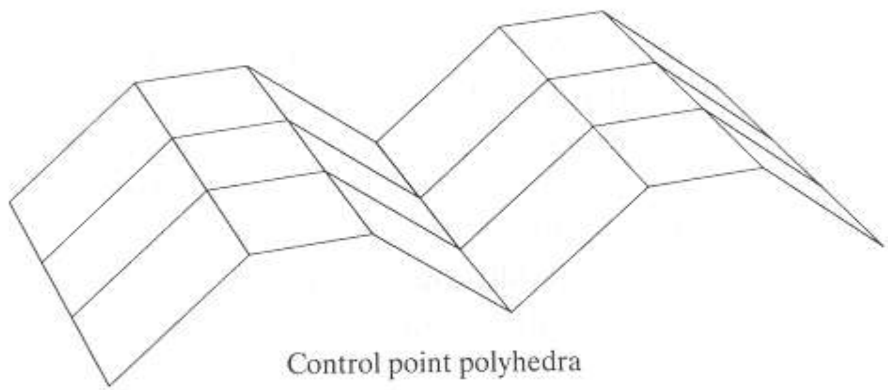
- Continuity constraints are similar to the ones for Bezier splines



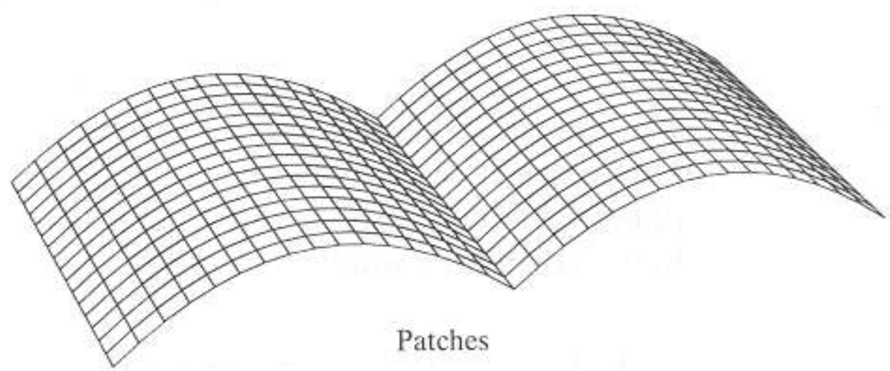
FvDFH Figure 11.43

Bezier Surfaces

- C^0 continuity requires aligning boundary curves



Control point polyhedra

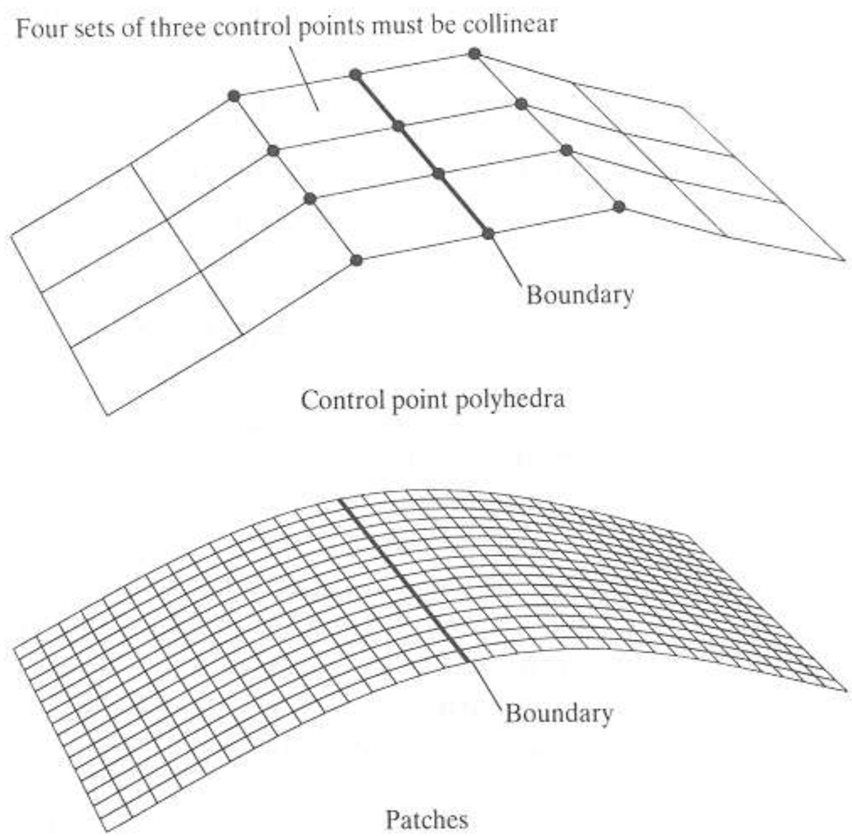


Patches

Watt Figure 6.26a

Bezier Surfaces

- C^1 continuity requires aligning boundary curves and derivatives



Watt Figure 6.26b

Parametric Surfaces

- Advantages:
  - Easy to enumerate points on surface
  - Possible to describe complex shapes
- Disadvantages:
  - Control mesh must be quadrilaterals
  - Continuity constraints difficult to maintain
  - Hard to find intersections

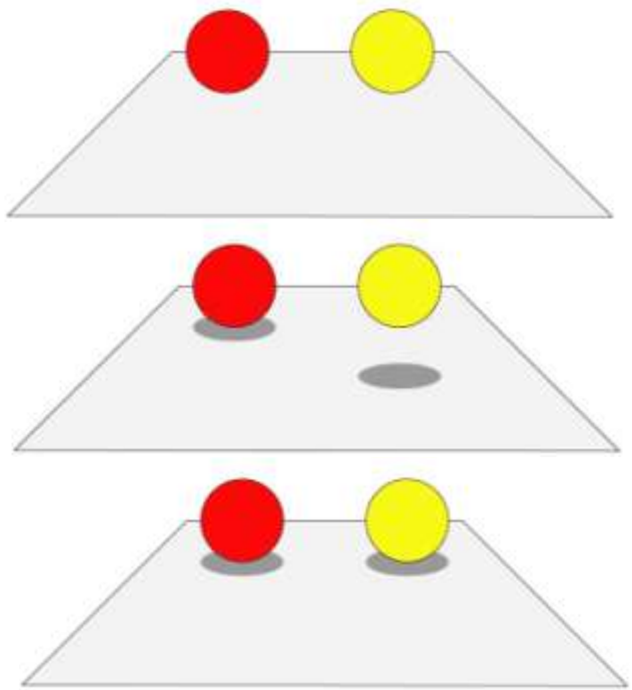
Comparison of Surface Reps

Feature	Polygonal Mesh	Parametric Surface	Subdivision Surface
Accurate	No	Yes	Yes
Concise	No	Yes	Yes
Intuitive specification	No	Yes	No
Local support	Yes	Yes	Yes
Affine invariant	Yes	Yes	Yes
Arbitrary topology	Yes	No	Yes
Guaranteed continuity	No	Yes	Yes
Natural parameterization	No	Yes	No
Efficient display	Yes	Yes	Yes
Efficient intersections	No	No	No

Rendering Leftovers

SHADOWS

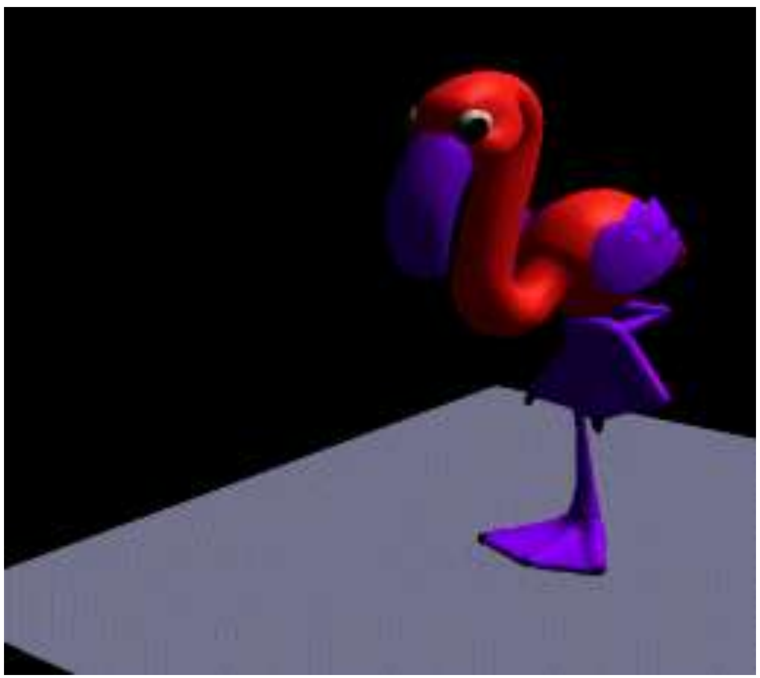
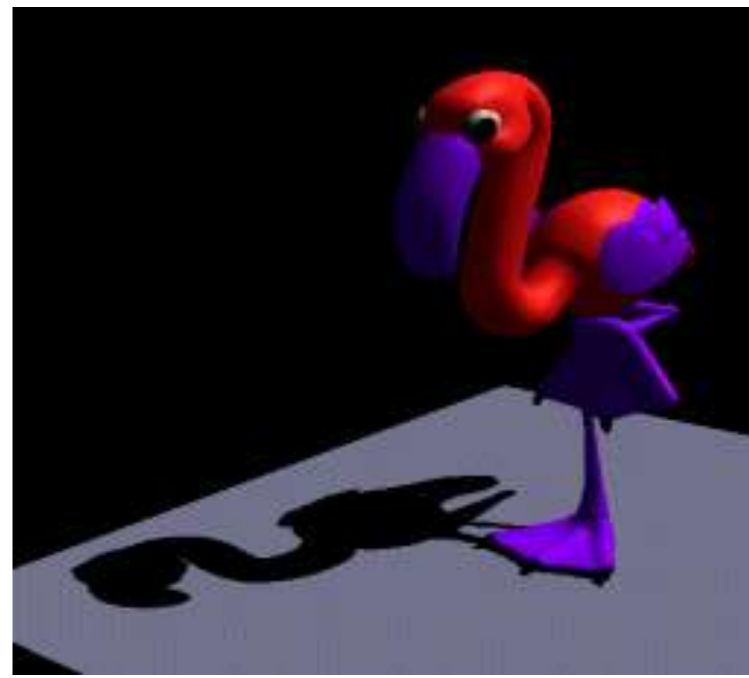
על חשיבותם של צללים



Trapezoid?

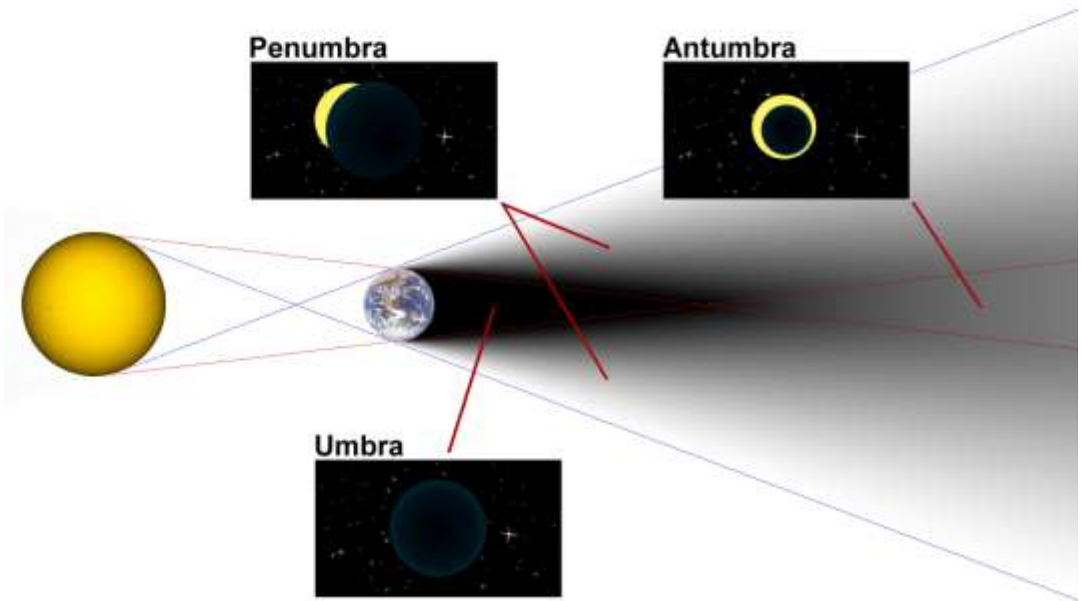
על חשיבותם של צללים

- Cue to object-object relationship.
- Provides additional depth cue.

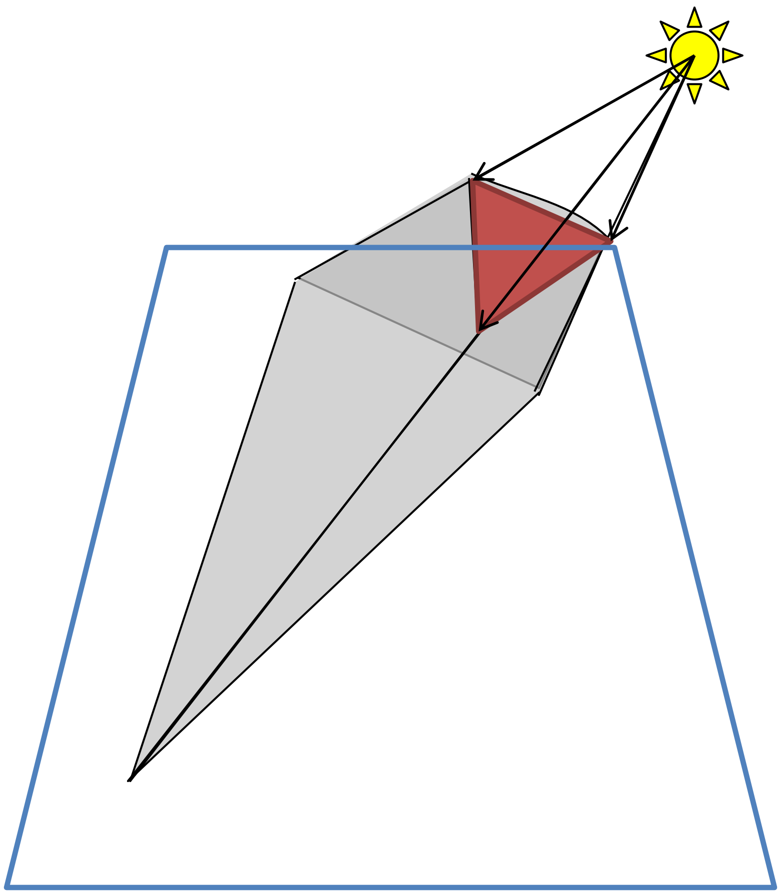
הגדרה

- צל – איזור בו אור המגיע ישירות ממקור אור לא מגיע בשל עצם מסתיר. הצללית שנוצרת היא של העצם החוסם

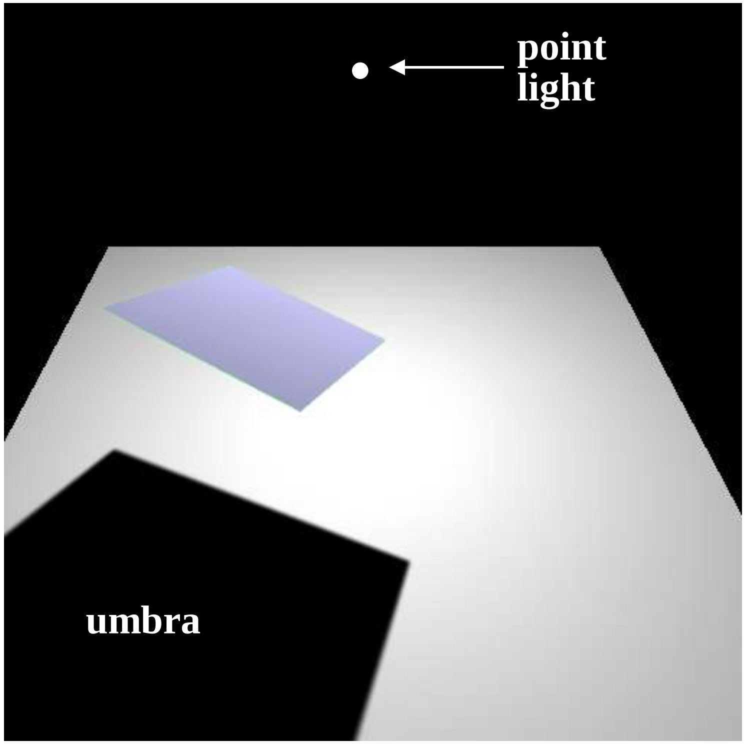


הגדרה: Shadow Volume

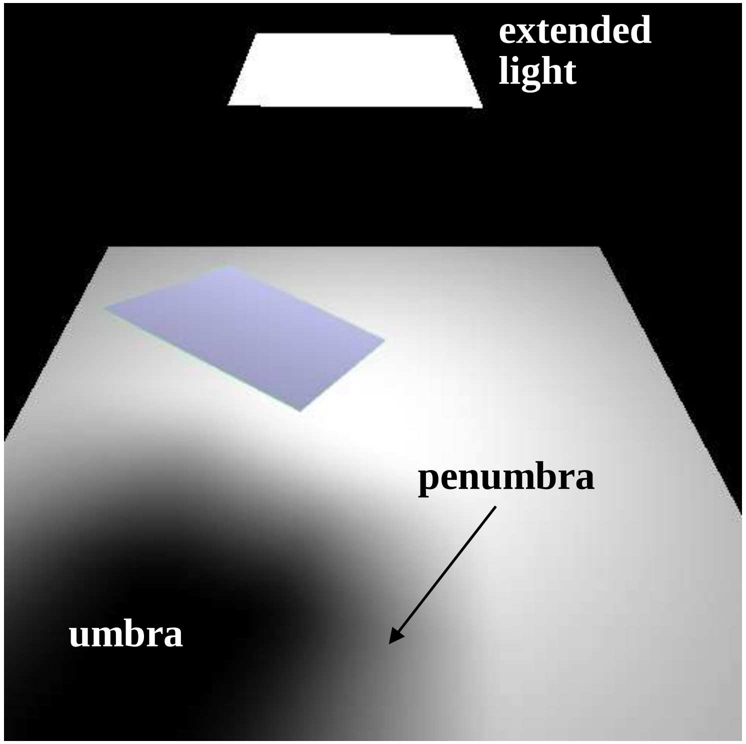
- Volume formed by extruding the occluder from the light source.
- Open and infinite
- Space inside the volume is in shadow.
- Space outside the volume is not.



צללים קשים ורכים



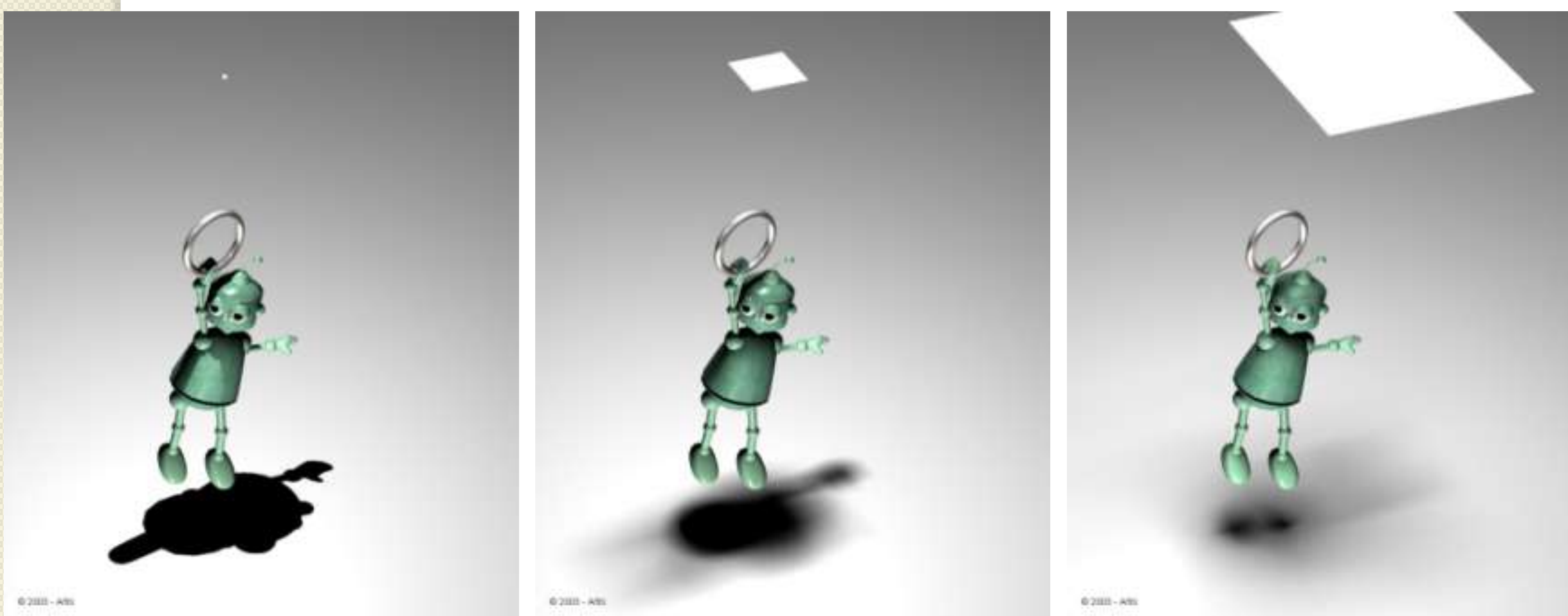
Hard shadow



Soft shadow

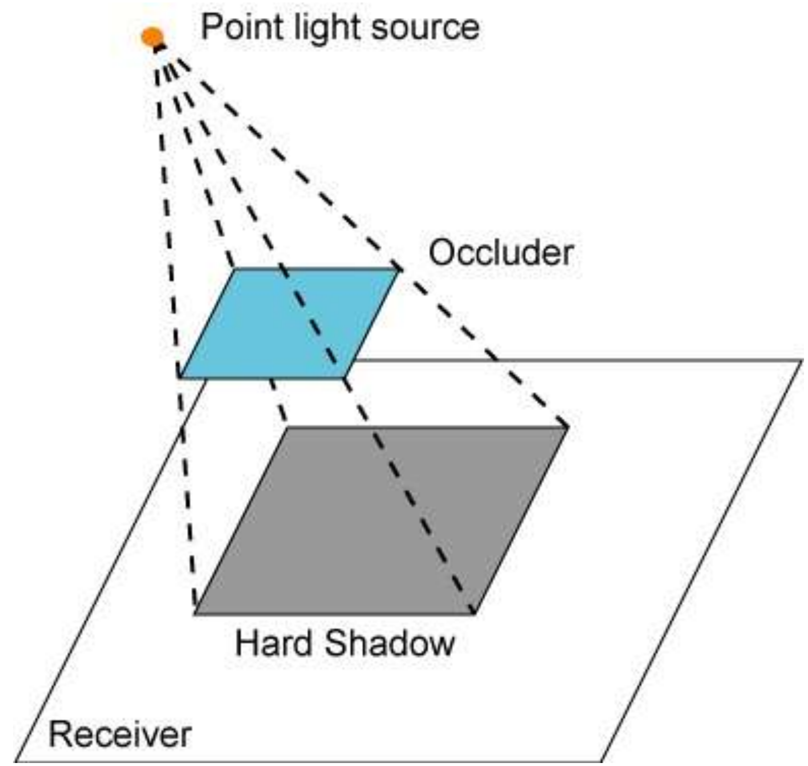
צללים רכים

- צל הוא פונקציה של גודל מקור האור, והמרחק שלו מהאובייקט

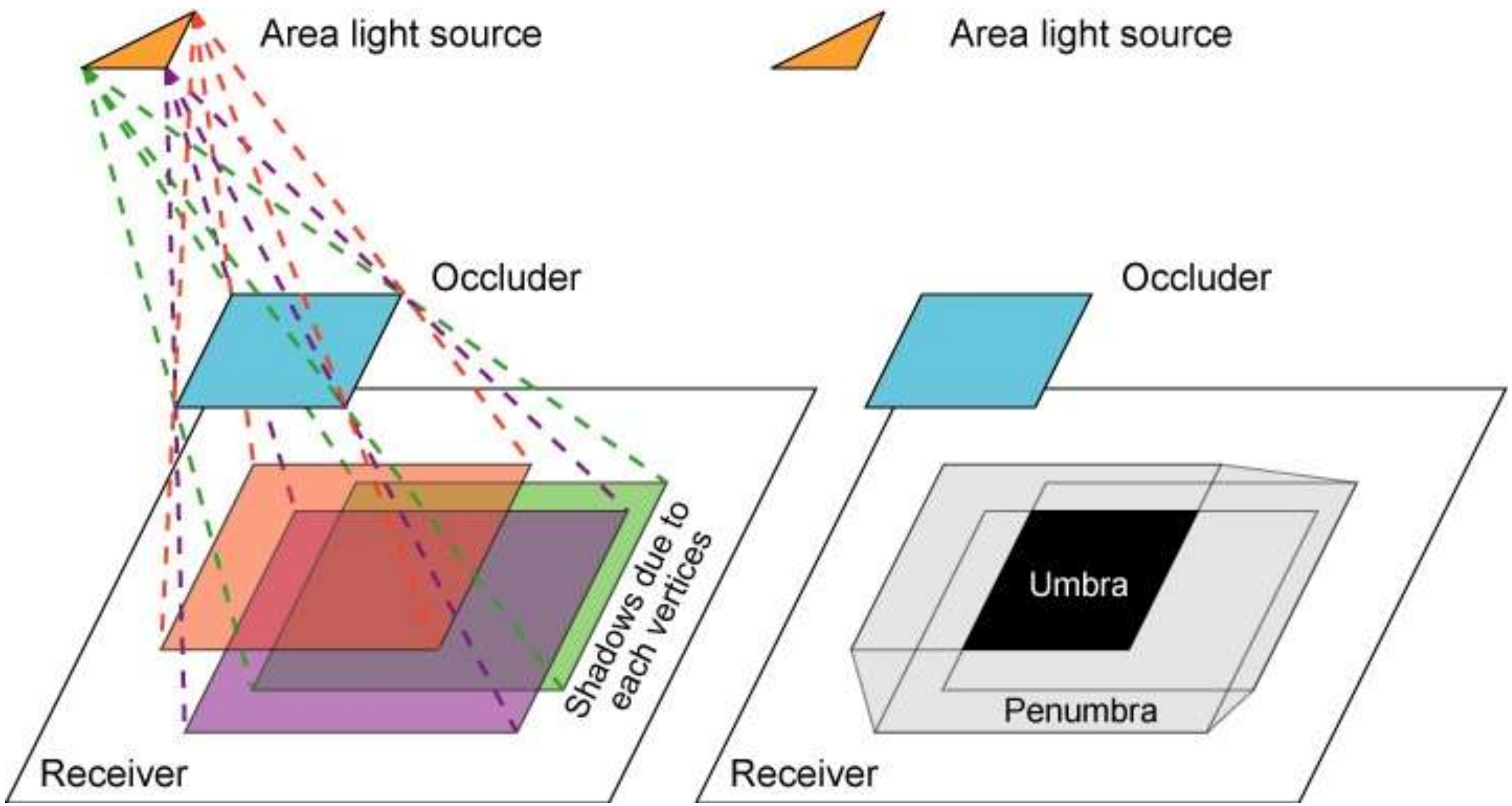


יצירת צללים קשים

- For every pixel light source is either visible or occluded



יצירת צללים רכים



מה משפיע על חישוב הצל?

- רמת הסיבוכיות של הסצינה
  - מספר מקורות האור
  - סוג מקורות האור
  - מספר אובייקטים מצלילים (Occluders)
  - מספר אובייקטים מקבלים (Receivers)
  - מיקום, גודל ועוצמת מקורות האור
- סטטי/דינמי
  - אובייקטים
  - תאורה
- הצללה עצמית
- שקיפות של עצמים
- דיוק וריאליזם של צללים

כיצד נחשב צל?

- מס' השיטות עצום ומספיק למלא קורס שלם
- נתמקד בכמה שיטות בסיסיות
- מהפשוט למסובך:

בלי צללים

צללים קשים

צללים רכים

אפקטים מיוחדים

צללים קשים (hard)

מקורות אור נקודתיים/כיווניים

- זיוף טוטאלי – מהיר אבל בד"כ לא מספיק טוב
  - Billboard
  - Projected object
- תמיכה בחומרה
  - Shadow textures
  - Shadow volumes
- חישוב מראש – מניח סטטיות (לפחות חלקית בסצינה
- Ray tracing – איטי מדי

צללים רכים

מקורות אור אזוריים

- תמיכה בחומרה – בד"כ מתייחסים לאור אזורי כאוסף של מקורות אור נקודתיים
  - שימוש ב-Accumulation buffer
  - Shadow volumes
  - Shadow textures
- חישוב מראש
  - Light maps
  - Discontinuity meshing
- Radiosity
- Ray-based

Shadows and OpenGL

- In OpenGL we send the geometry for a model through the pipeline.
- The Visibility function, ***V***, is not a constant in our illumination model.
  - Per vertex information?
  - Per fragment using a texture map?
  - Some per-pixel masking function?
- Recall that we need a ***V*** for each light.

Masking in OpenGL

- OpenGL provides several ways of *masking* pixels
  - Stencil buffer with stencil test
  - Alpha test with fragment's alpha values
  - Blending with fragment's and framebuffer's alpha values.
  - Texture sampling and shaders.

Positive Light

- Algorithm
  - Render scene with ambient illumination only
  - For each light source
    - Render scene with illumination from this light only
    - Scale illumination by shadow mask
    - Add contribution to frame buffer

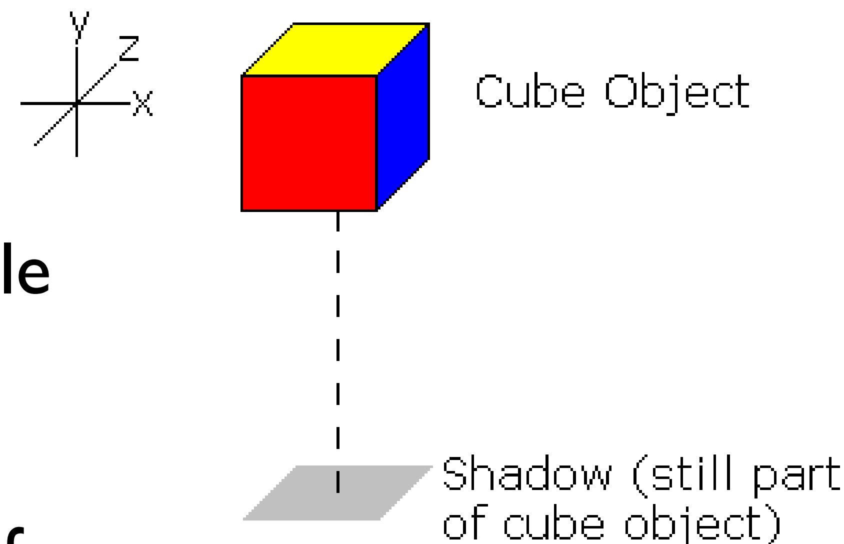
Ad-Hoc and Custom Shadows

פתרון פשוט יהיה פשוט לזייף צללים

- Fake proxy geometry.
- Projection of model to a plane.
- Projection of a texture to a plane.

Fake Proxy Geometry

- Approximation of shadow position and shape based on object's typical use.
- Typically assumes overhead lighting.
- Typically assumes a single flat ground plane as a receiver.
- E.g., draw the bottom of the bounding box.

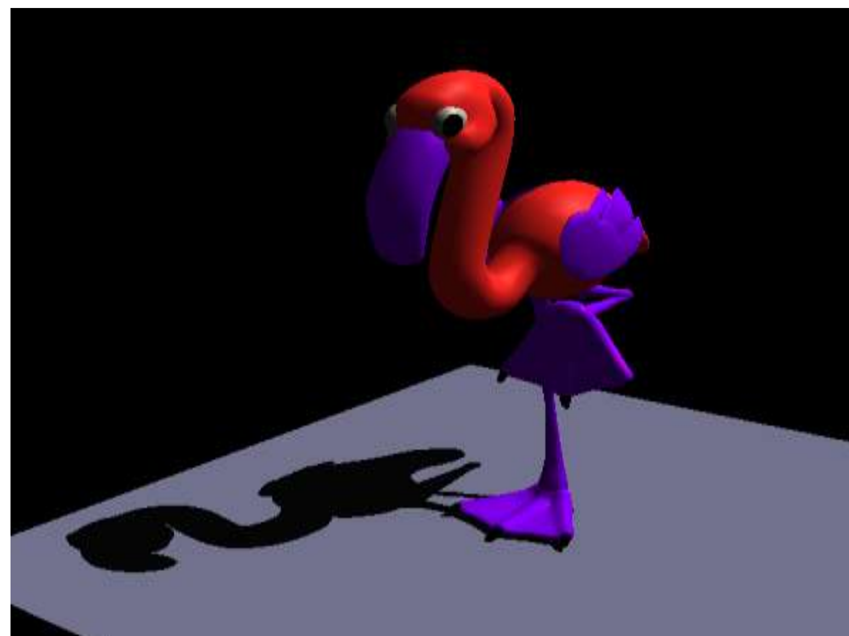


Fake Proxy Geometry

- Quite complex model.
- Know it will sit on a flat floor.
- Will fail if we place another object behind or underneath it.

Projected Geometry

- [Blinn88] *Me and my fake shadow*
  - Shadows for selected large receiver polygons
    - Ground plane
    - Walls

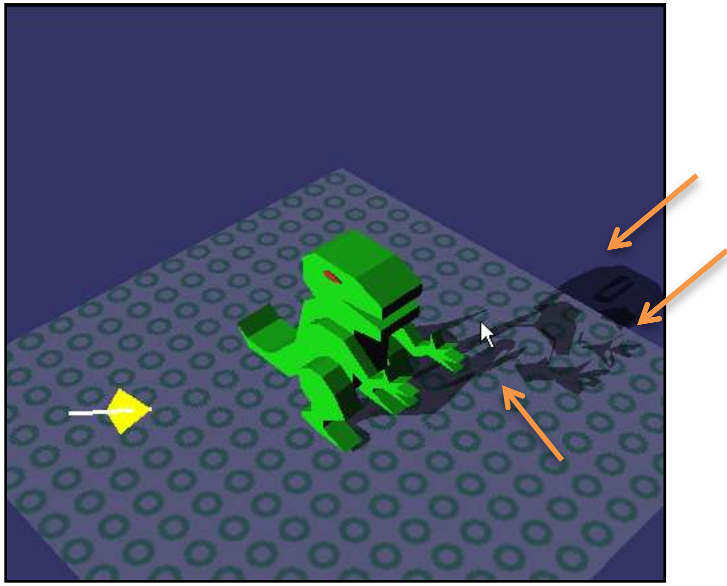


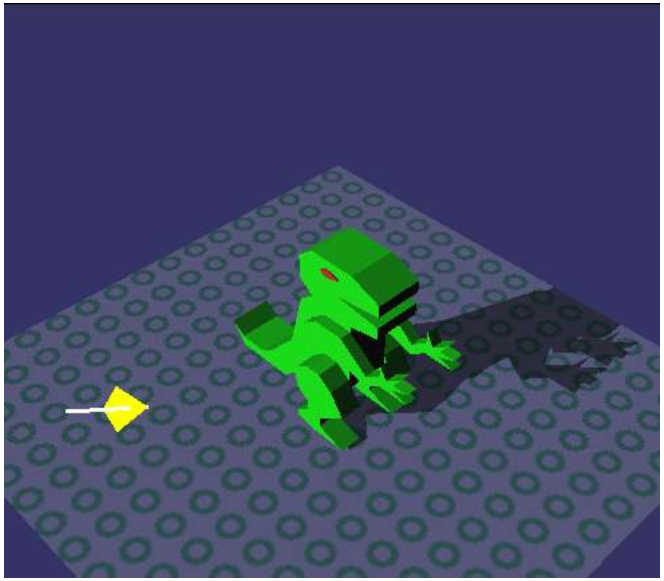
Projected Geometry

- Basic algorithm
  - Render scene (full lighting)
  - For each receiver polygon
    - Compute projection matrix *M*
    - Multiply with actual transformation (modelview)
    - Render selected (occluder) geometry
      - Darken/Black

Projected Geometry Problems

- Z-Fighting
  - Use bias when rendering shadow polygons
  - Use stencil buffer (no depth test)
- Bounded receiver polygon
  - Use stencil buffer (restrict drawing to receiver area)
- Shadow polygon overlap
  - Use stencil count (only the first pixel gets through)





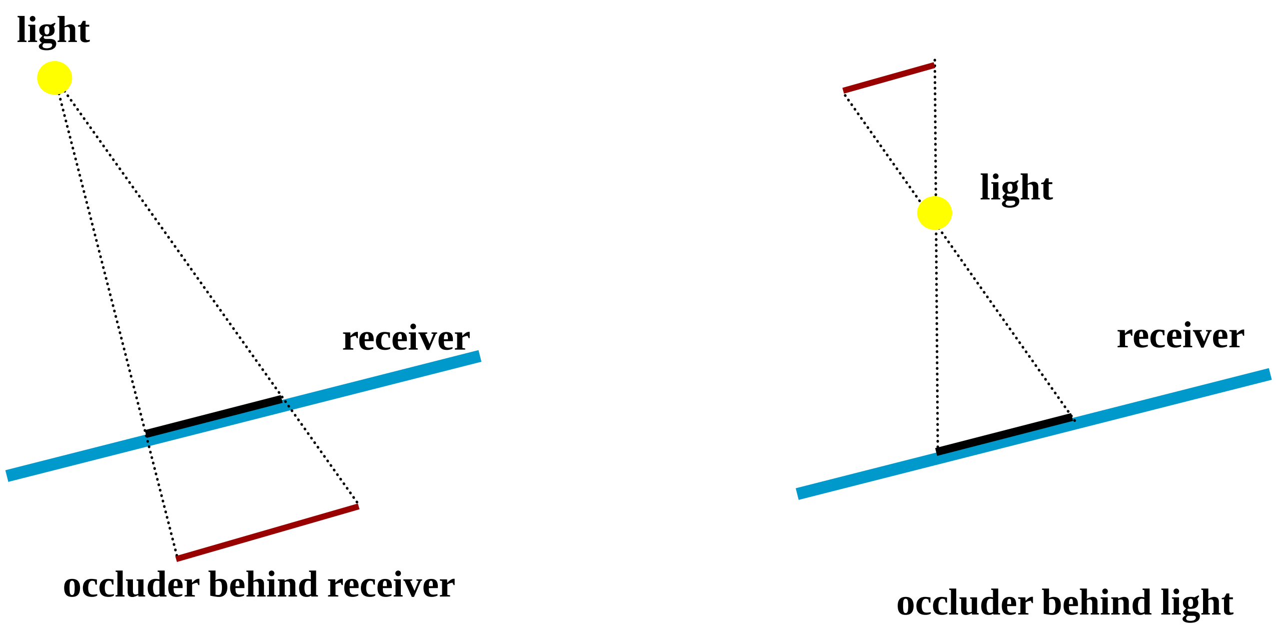
Projected Geometry Algorithm

- Stencil buffer algorithm (1bit stencil)

```
1. Render scene without receiver polygon
2. Clear stencil buffer
3. Render receiver polygon
   - stencil operation ‘set’ (visible pixels)
4. Render shadow polygons
   - without depth test
   - stencil test ‘is set?’
   - stencil operation ‘clear’
   - blending e.g. ‘dest = dest * 0.2’
(darken)
```

Projected Geometry Problems

- Wrong Shadows & Anti-Shadows
  - Objects behind light source
  - Objects behind receiver



Projected Geometry

- Summary
  - Only practical for very few, large receivers
  - Easy to implement
  - Use stencil buffer (z fighting, overlap, receiver)
  - Efficiency can be improved by rendering shadow polygons to texture maps
    - Occluders and receiver 'static' for some time

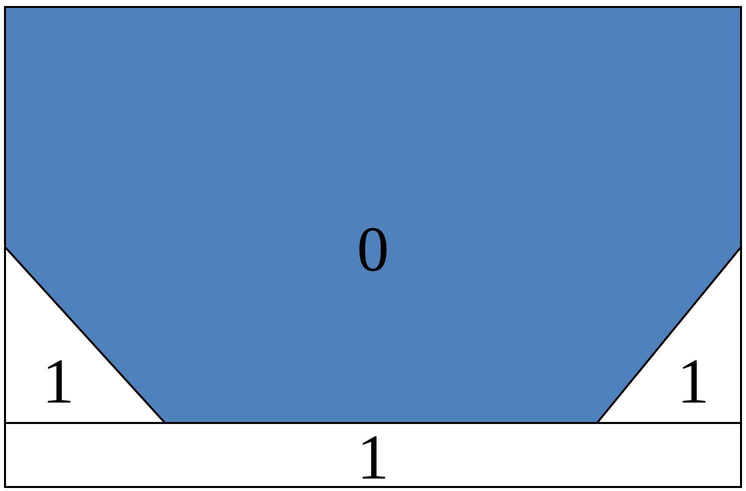
STENCIL BUFFER

Stencil Buffer

- The Stencil Buffer is another frame buffer, like the Color Buffer, Depth Buffer and Accumulation Buffer.
- Stencil Buffer can be used to specify a pattern so that only fragments that pass the stencil test are rendered to the color buffer.

Stencil Buffer Example

Render the fragments only where the stencil buffer bit is 0



Stencil Buffer

Color Buffer

שימוש בStencil buffer -

- אתחול

```
glClearStencil(0);
glClear(GL_COLOR_BUFFER_BIT|GL_STENCIL_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glEnable(GL_STENCIL_TEST);
```

- הגדרת המבחן

```
glStencilFunc(GL_EQUAL,    //comparison function
         0x1,              // reference value
         0xff);            // comparison mask
```

- התוצאה תלויה במבחן הזה + מבחן עומק (3 תוצאות אפשריות). התוכניתן קובע מה עושים בכל מקרה (שמור, החלף, אפס, קדם ב-1, חסר 1, הפוך

```
glStencilOp(GL_KEEP, GL_DECR, GL_INCR);
glStencilMask(0xff);
```

OpenGL Stencil Buffer Functions

```
// glStencilFunc: set function and reference value for stencil testing
// func :Specifies the test function. Options: GL_NEVER, GL_LESS, GL_LEQUAL,
//     GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, and GL_ALWAYS.
//     Default is GL_ALWAYS.
// ref : Specifies the reference value for the stencil test.
//     ref is clamped to the range [0,2n−1], where n is the number of bits for each fragment
//     in the stencil buffer. The initial value is 0.
// Mask: Specifies a mask that is ANDed with both the reference value and the stored
//     stencil value when the test is done. Default is all 1's.
void glStencilFunc (GLenum func , GLint ref , GLuint mask );
```

```
// glStencilOp: set stencil test actions on the stencil buffer
 //fail:Specifies the action to take when the stencil test fails. Options: GL_KEEP, GL_ZERO ,
    //GL_REPLACE, GL_INCR, GL_DECR, and GL_INVERT. Default is GL_KEEP .
 //zfail: Specifies the stencil action when the stencil test passes, but the depth test fails .
    //Options and default same as for fail.
 //zpass: Specifies the stencil action when both the stencil test and the depth test pass ,
    //or when the stencil test passes and either there is no depth buffer
    //or depth testing is not enabled. Options and default same as for fail .
void glStencilOp (GLenum fail , GLenum zfail , GLenum zpass );
```

עוד דוגמה לאתחול ושימוש...

- First, make sure we request the stencil buffer.

```
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB |
                     GLUT_DEPTH | GLUT_STENCIL );
```

- Next, make sure we enable stencil test

```
glEnable(GL_STENCIL_TEST);
```

- Example: Make a stencil buffer have value 1 inside a diamond shape and value 0 outside.

```
glClearStencil(0x0); // specify stencil clear value
glClear(GL_STENCIL_BUFFER_BIT); // clear stencil buffer
// Set the ref value to 0x1
glStencilFunc(GL_ALWAYS, 0x1, 0x1);
// Replace stencil bit with ref (0x1) whenever we process a fragment
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
// draw a diamond (we'll not really render the color buffer,
// but just use this to set the stencil buffer)
glBegin(GL_QUADS);
glVertex2f(-1,0);
glVertex2f(0,1);
glVertex2f(1,0);
glVertex2f(0,-1);
glEnd();
```

How to use Stencil Buffer to filter rendering to Color Buffer

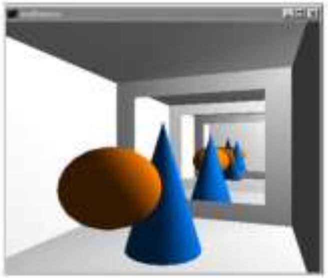
```
// render scene only where stencil buffer is 0
void display() {

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

 // fragment passes the test if stencil value at fragment is not equal to 0x1
 glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);

 // don’t change the value of the stencil buffer in any case
 glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

 // render the scene
 renderScene();
}
```



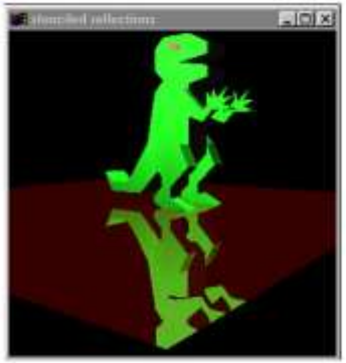
שימוש ב-Stencil Buffer

- השתקפויות
  - רנדר את הסצנה כרגיל
  - לכל מראה

```
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, ~0);
glColorMask(0,0,0,0);
renderMirrorSurfacePolygons(thisMirror);

glDepthRange(1,1);                      // always
glDepthFunc(GL_ALWAYS);                 //  write the farthest depth value
glStencilFunc(GL_EQUAL, 1, ~0);         // match mirror's visible pixels
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // do not change stencil values
renderMirrorSurfacePolygons(thisMirror);
```

  - כעת נרנדר את ההשתקפות עצמה



שימוש ב-Stencil Buffer

◦ נרנדר את ההשתקפות

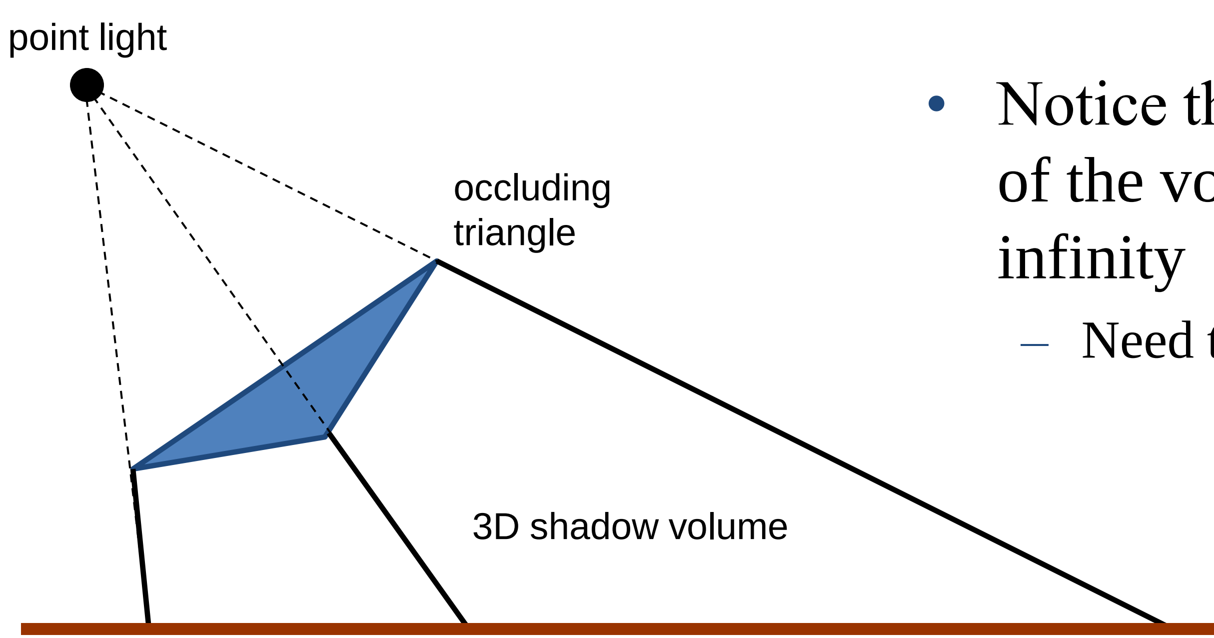
```
GLfloat matrix[4][4];
GLdouble clipPlane[4];
glPushMatrix();
  // returns world-space plane equation for mirror plane to use as clip plane
  computeMirrorClipPlane(thisMirror, &clipPlane[0]);
  // set clip plane equation
  glClipPlane(GL_CLIP_PLANE0, &clipPlane);
  // returns mirrorMatrix for given mirror
  computeReflectionMatrixForMirror(thisMirror, &matrix[0][0]);
  // concatenate reflection transform into modelview matrix
  glMultMatrixf(&matrix[0][0]);
  glCullFace(GL_FRONT);
  drawScene();                                   // draw everything except mirrors
  drawOtherMirrorsAsGraySurfaces(thisMirror);  // draw other mirrors as
                                                 // neutral "gray" surfaces
  glCullFace(GL_BACK);
  glDisable(GL_CLIP_PLANE0);
glPopMatrix();
```

SHADOW VOLUMES

Doom 3

Shadow Volumes

- A volume of space formed by an occluder
- Bounded by the edges of the occluder

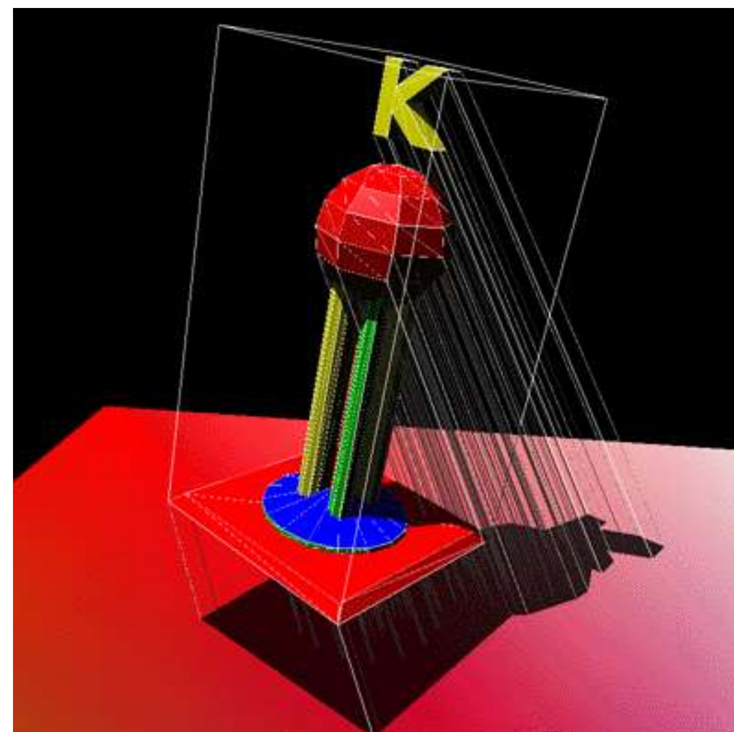


- Notice that the "far" end of the volume goes to infinity
  - Need to cap it

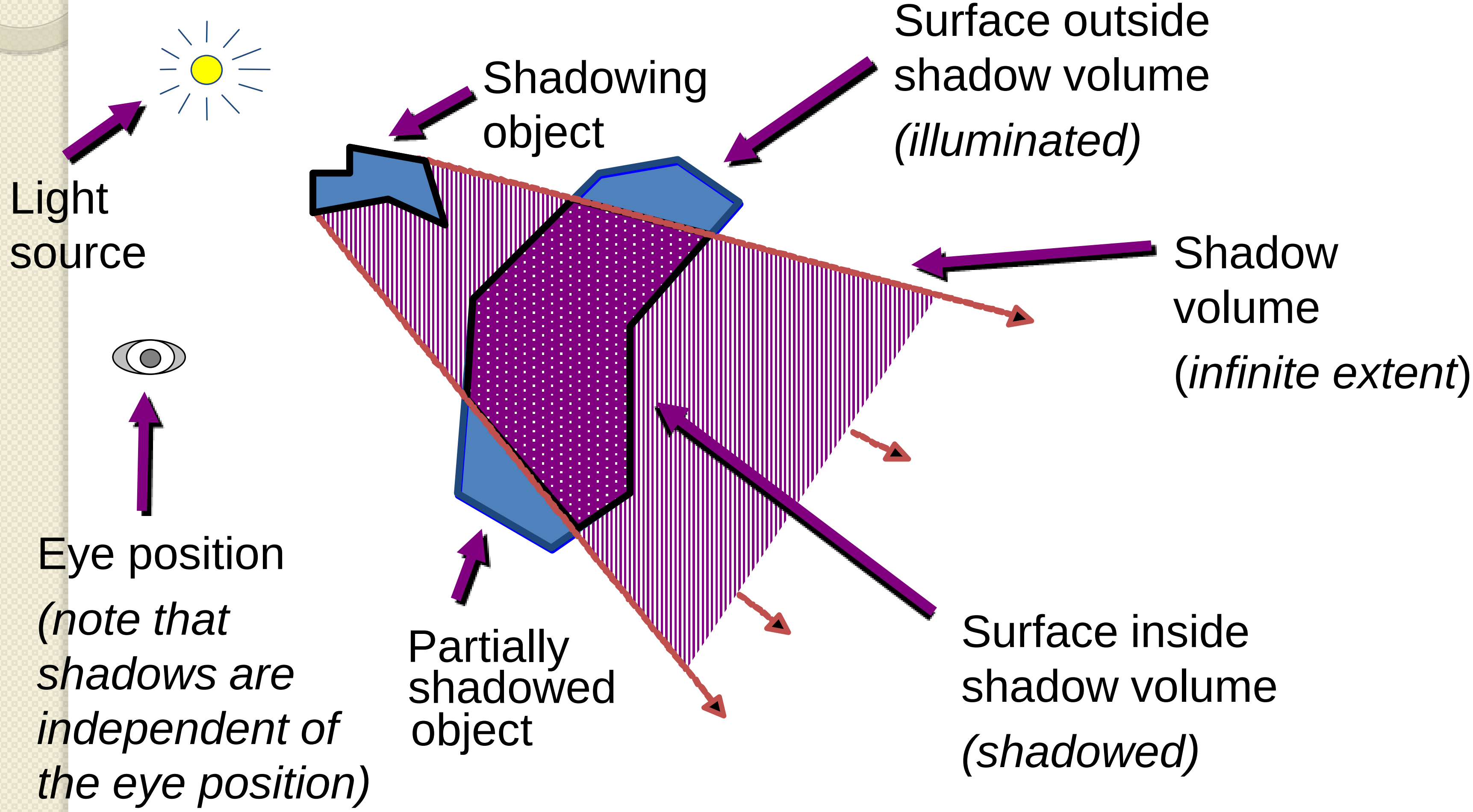
Shadow Volumes

- Compute shadow volume for all visible polygons from the light source

- Add the shadow volume polygons to your scene database
  - Tag them as shadow polygons
  - Assign its associated light source



2D Cutaway of a Shadow Volume



Shadow Volume Advantages

- Omni-directional approach
  - Not just spotlight frustums as with shadow maps
- Automatic self-shadowing
  - Everything can shadow everything, including self
  - Without *shadow acne* artifacts as with shadow maps
- Window-space shadow determination
  - Shadows accurate to a pixel (Object method)
  - Or sub-pixel if multisampling is available
- Required stencil buffer broadly supported today
  - OpenGL support since version 1.0 (1991)
  - Direct3D support since DX6 (1998)

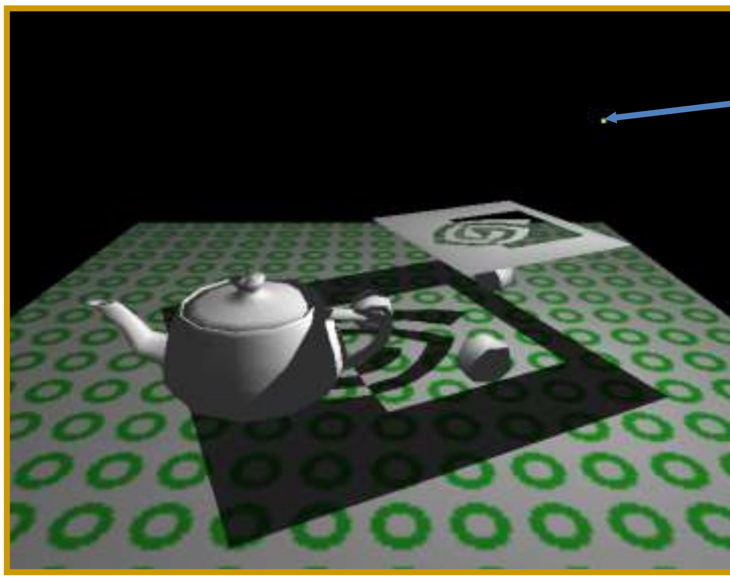
Shadow Volume Disadvantages

- Ideal light sources only
  - Limited to local point and directional lights
  - No area light sources for soft shadows
- Requires polygonal models with connectivity
  - Models must be closed (2-manifold)
  - Models must be free of non-planar polygons
- Silhouette computations are required
  - Can burden CPU
  - Particularly for dynamic scenes
- Inherently multi-pass algorithm
- Consumes lots of GPU fill rate

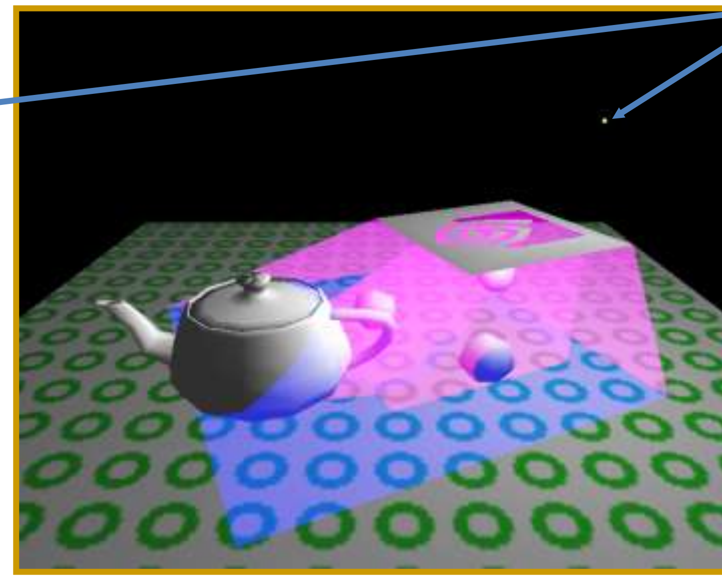
Visualizing Shadow Volumes in 3D

- Occluders and light source cast out a shadow volume
  - Objects within the volume should be shadowed

Light source



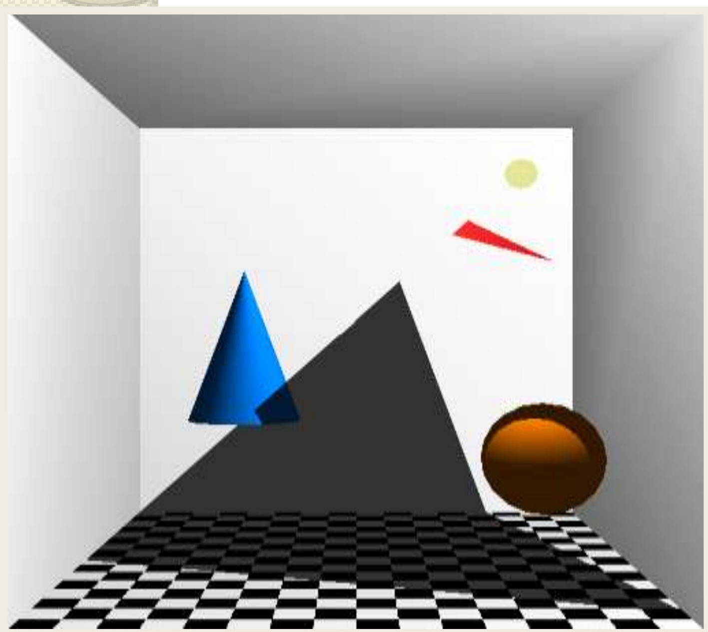
Scene with shadows from an NVIDIA logo casting a shadow volume



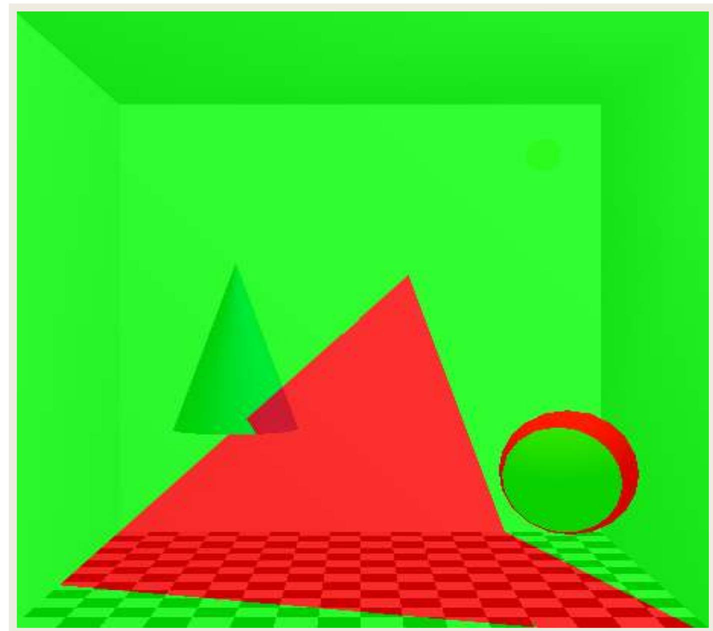
Visualization of the shadow volume

Visualizing the Stencil Buffer Counts

Shadowed scene



Stencil buffer contents



Stencil counts beyond 1 are possible for multiple or complex occluders.

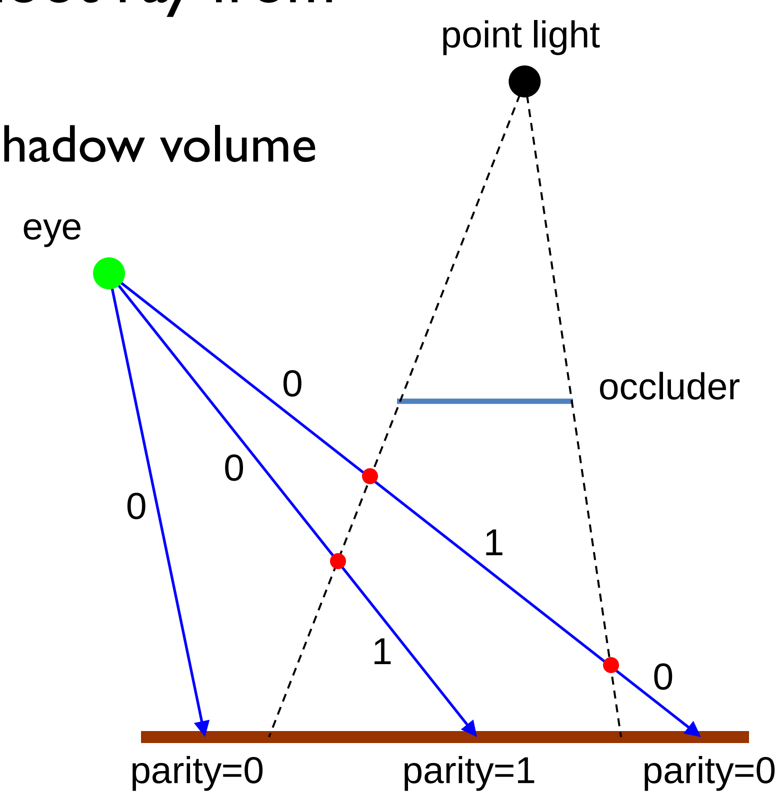
red = stencil value of 1
green = stencil value of 0

Shadow Volumes

When is a surface point inside shadow?

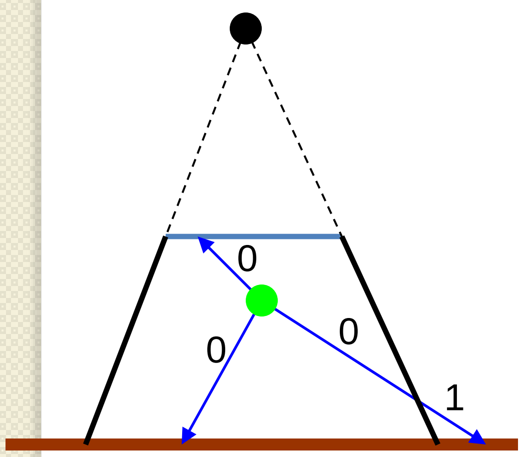
- Use a parity test similar to a "ray inside-outside" test
- Initially set parity to 0 and shoot ray from eye to P
  - Invert parity when ray crosses shadow volume boundary
  - parity = 0, not in shadow,

    parity = 1, in shadow

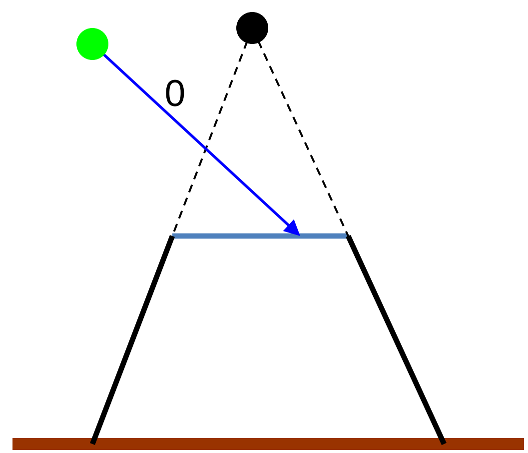


Problems With Parity Test

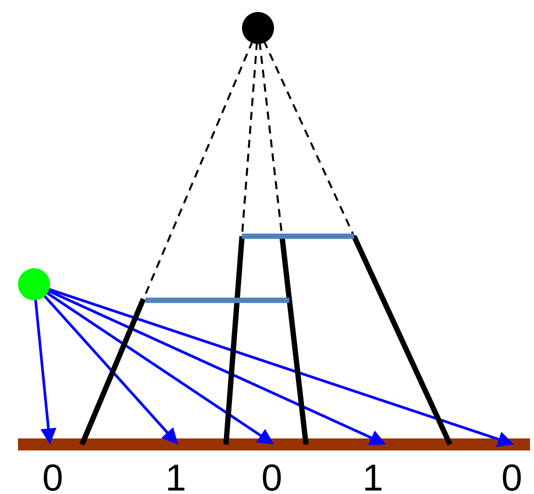
Eye inside of shadow volume



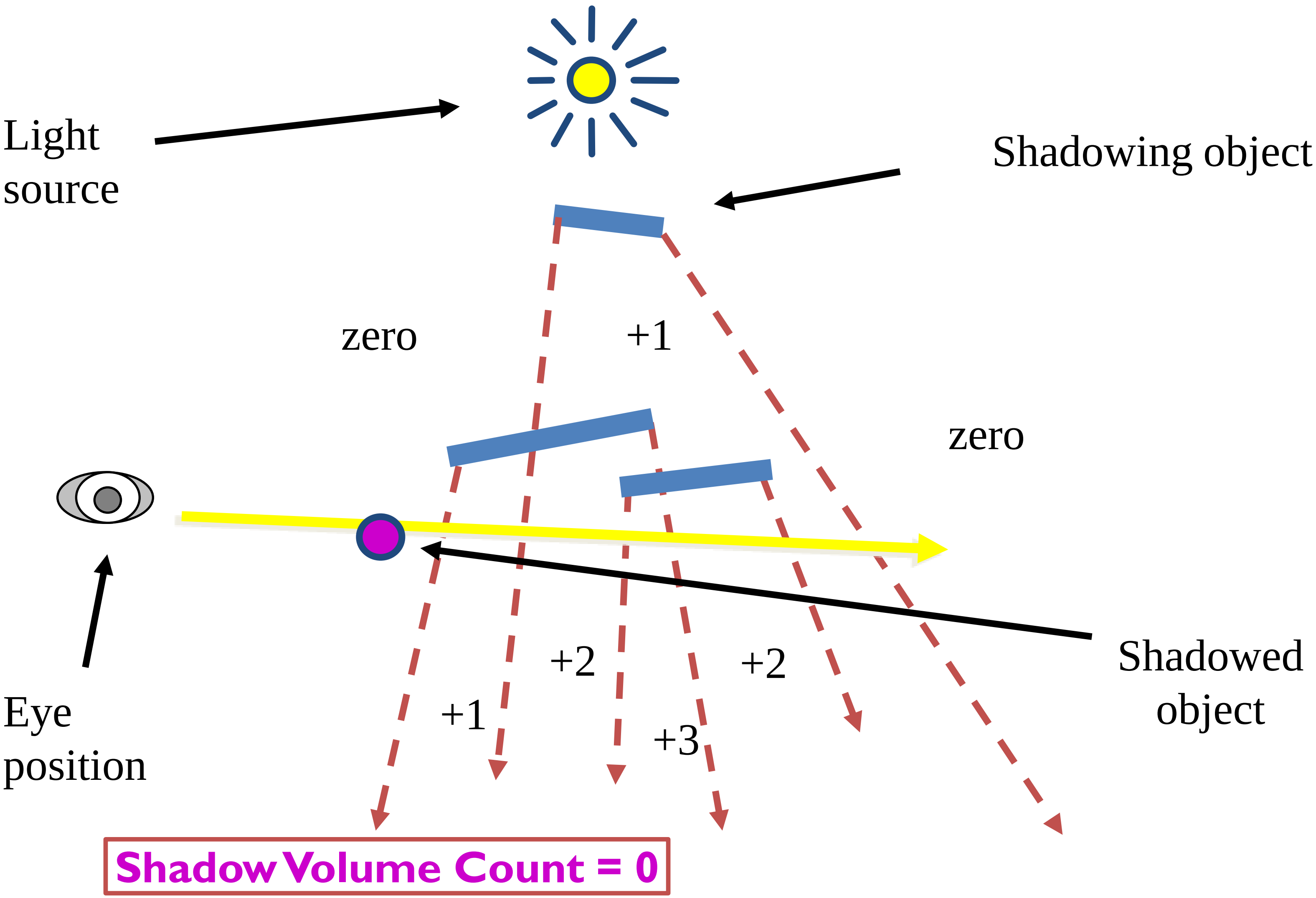
Self-shadowing of visible occluders



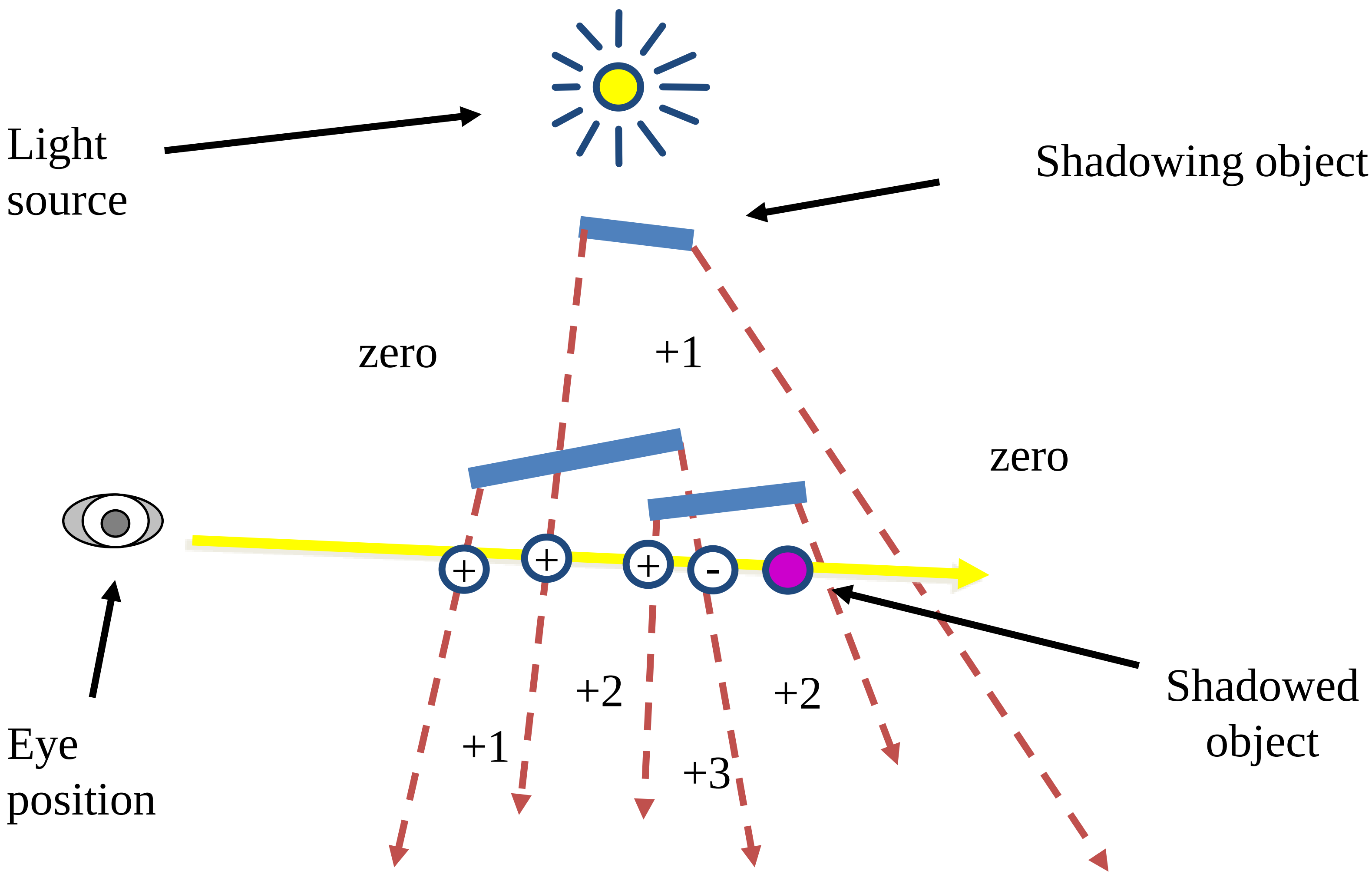
Multiple overlapping shadow volumes



Better Solution : Counter

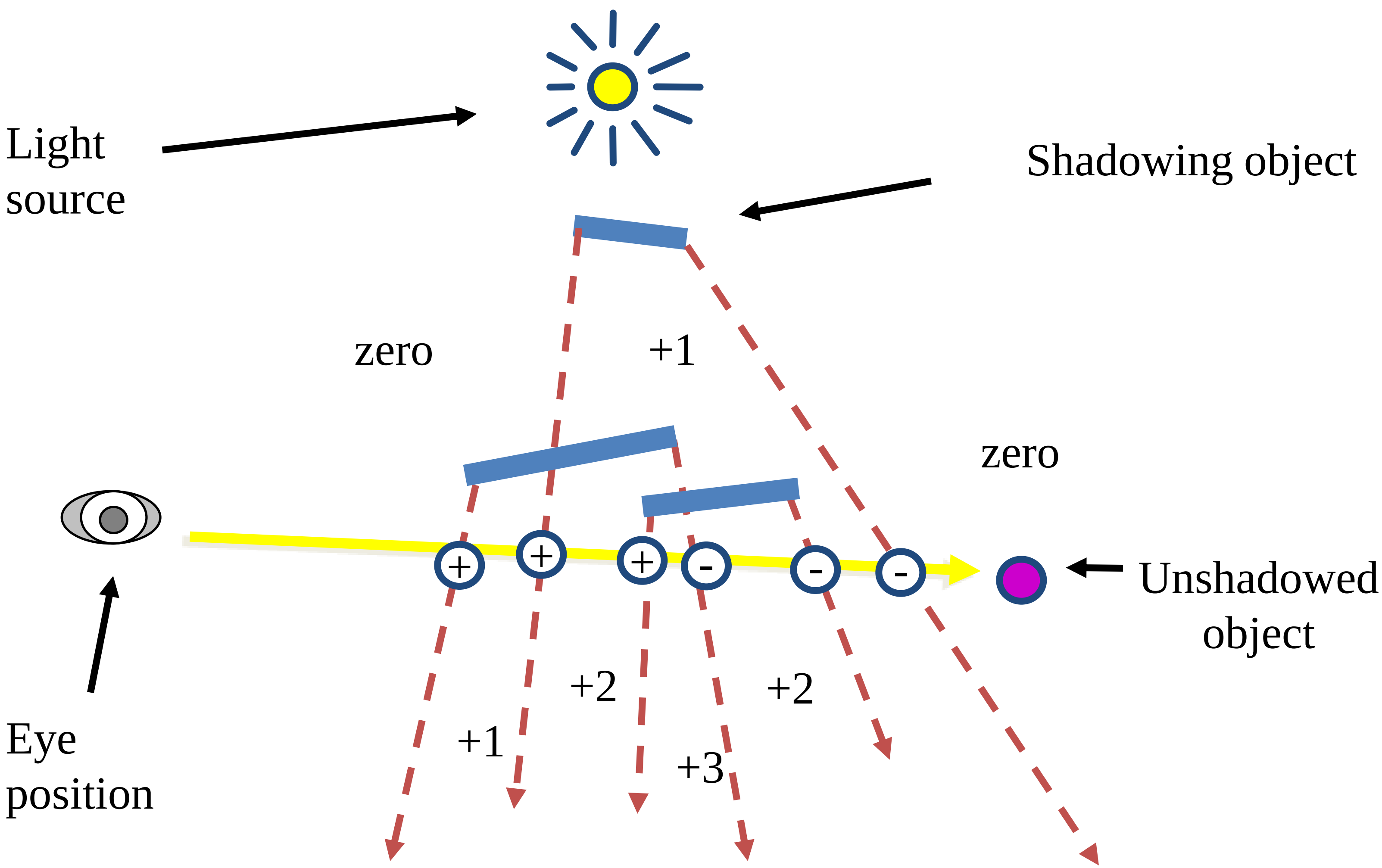


Better Solution : Counter



Shadow Volume Count = +1+1+1-1 = 2

Better Solution : Counter

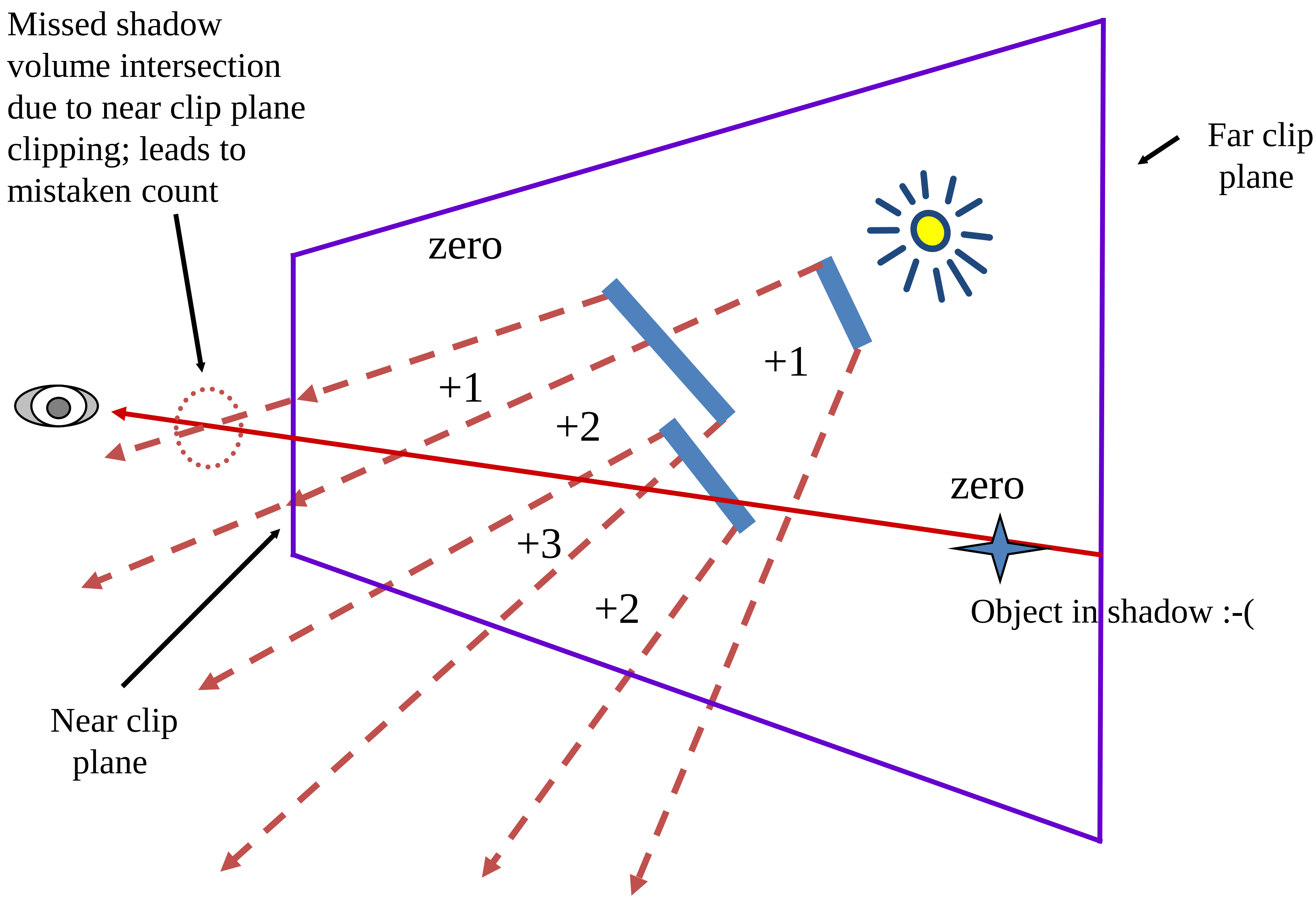


Shadow Volume Count = +1+1+1-1-1-1 = 0

Graphics Hardware Approach Using The Stencil Buffer

- *Zpass approach*
  - Render visible scene to depth buffer
  - Turn off depth and color, turn on stencil
  - Init. stencil buffer given viewpoint
  - Draw shadow volume twice using face culling
    - 1st pass: render *<u>front</u>* faces and *<u>increment</u>* when depth test passes
    - 2nd pass: render *<u>back</u>* faces and *<u>decrement</u>* when depth test passes
- stencil pixels != 0 in shadow, = 0 are lit

Zpass Problem



Zfail Approach

- Render visible scene to depth buffer
- Turn off depth and color, turn on stencil
- Init. stencil buffer given viewpoint
- Draw shadow volume twice using face culling
  - 1st pass: render *back* faces and *increment* when depth test fails
  - 2nd pass: render *front* faces and *decrement* when depth test fails
- stencil pixels != 0 in shadow, = 0 are lit

Clipping Plane Problem

- *Zpass* : Near clipping plane
  - Move near clipping plane closer to eye?
    - Lose depth precision in perspective
- *Zfail* : Far clipping plane
  - Move far clipping plane closer to eye?
    - Set far clipping plane to infinity.
    - See "***Practical & Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering***" by Cass Everitt & Mark J. Kilgard, Nvidia

Zfail versus Zpass Comparison (1)

- **When stencil increment/decrements occur**
  - ***Zpass:* on depth test pass**
  - ***Zfail:* on depth test fail**
- **Increment on**
  - ***Zpass:* front faces**
  - ***Zfail:* back faces**
- **Decrement on**
  - ***Zpass:* front faces**
  - ***Zfail:* back faces**

Zfail versus Zpass Comparison (2)

- Both cases order passes based stencil operation
  - First, render *increment* pass
  - Second, render *decrement* pass
  - Why?
    - Because standard stencil operations saturate
    - Wrapping stencil operations can avoid this
- Which clip plane creates a problem
  - *Zpass:* near clip plane
  - *Zfail:* far clip plane
- Either way is foiled by view frustum clipping
  - Which clip plane (front or back) changes

Insight!

- If we could avoid *either* near plane *or* far plane view frustum clipping, shadow volume rendering could be robust
- Avoiding near plane clipping
  - Not really possible
  - Objects can always be behind you
  - Moreover, depth precision in a perspective view goes to hell when the near plane is too near the eye
- Avoiding far plane clipping
  - Perspective make it possible to render at infinity
  - Depth precision is terrible at infinity, but we just care about avoiding clipping

Examples

Scene with shadows.
Yellow light is embedded in the green three-holed object. P_{inf} is used for all the following scenes.

Same scene visualizing the shadow volumes.

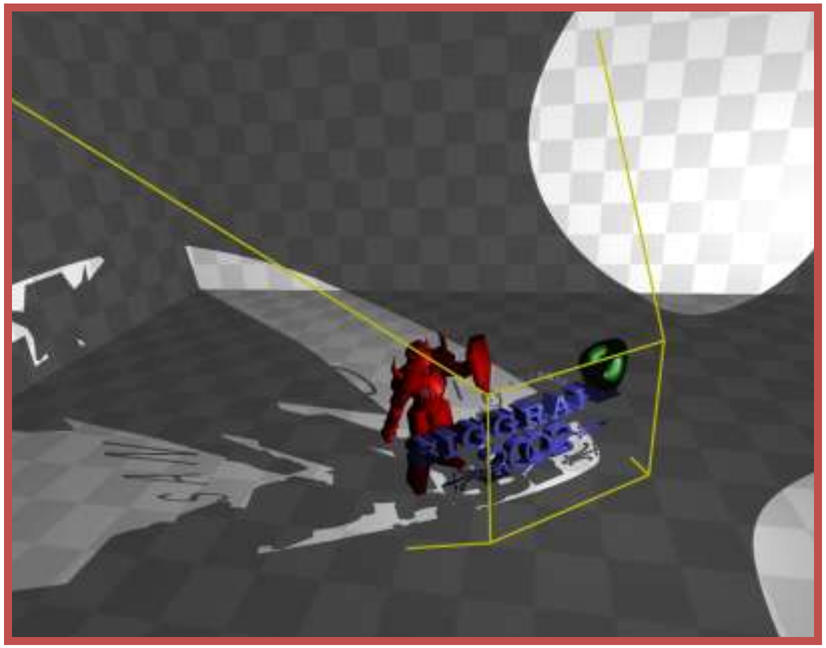
Examples

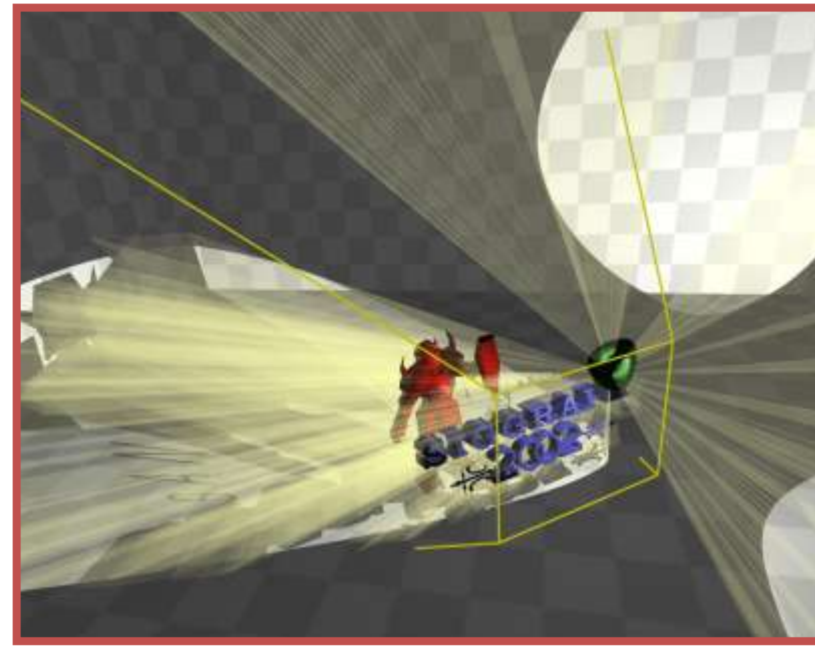
Details worth noting . . .

Fine details: Shadows of the A, N, and T letters on the knight's armor and shield.

Hard case: The shadow volume from the front-facing hole would definitely intersect the near clip plane.

Examples





Alternate view of same scene with shadows. Yellow lines indicate previous view's view frustum boundary. Recall shadows are view-independent.

Shadow volumes from the alternate view.

Stenciled Shadow Volumes with Multiple Lights

Three colored lights. Diffuse/specular bump mapped animated characters with shadows. 34 fps on GeForce4 Ti 4600; 80+ fps for one light.

Stenciled Shadow Volumes for Simulating Soft Shadows

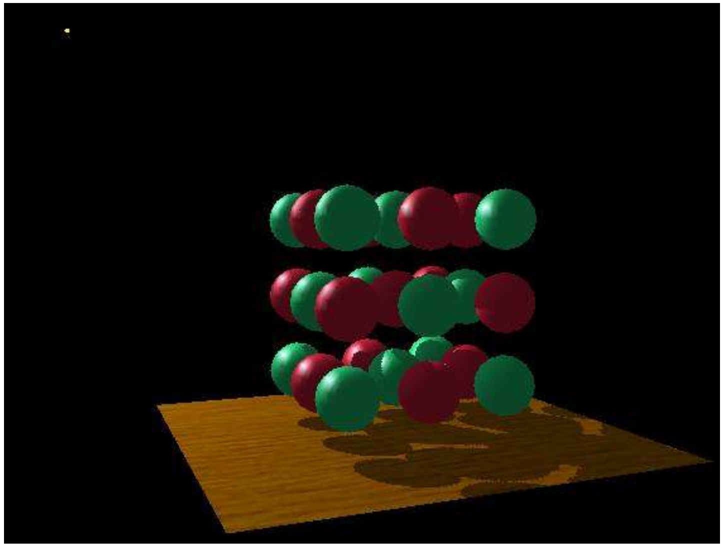
Cluster of 12 dim lights approximating an area light source. Generates a soft shadow effect; careful about banding. 8 fps on GeForce4 Ti 4600.

The cluster of point lights.

Issues With Shadow Volumes

- The addition of shadow volume polygons can greatly increase your database size
- Using the stencil buffer approach, pixel fill becomes a key speed factor
- Create a shadow volume from the silhouette of an object instead of each polygon
- Take care when coding the algorithm

SHADOW MAPS



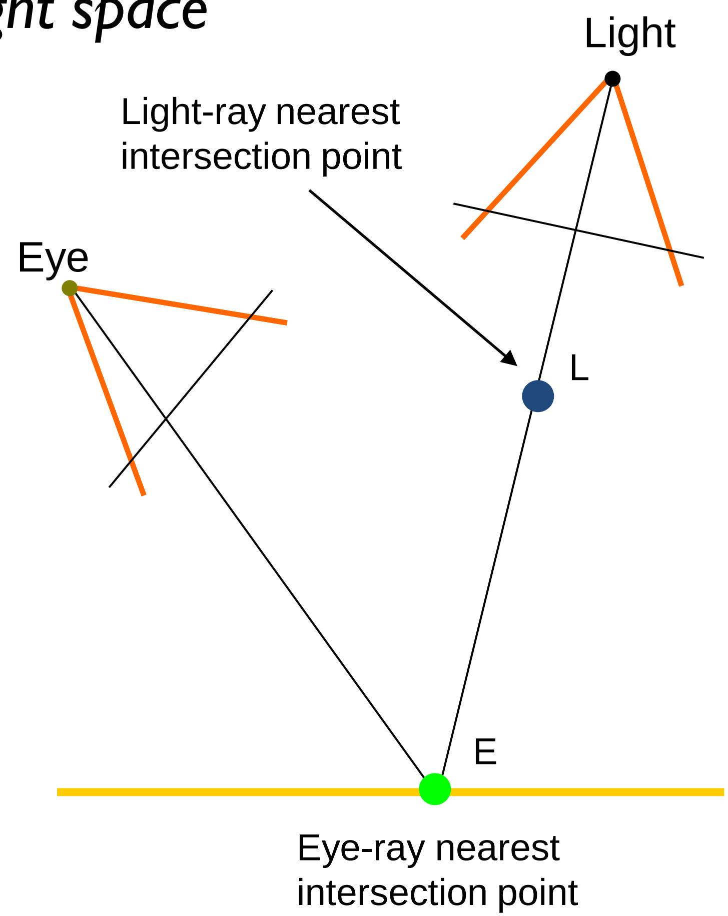
Z-Buffer Shadow Maps

- Define a coordinate system (*light space*) such that the light is the center of projection
- Render a depth buffer (*z-buffer*) of the visible scene, each pixel (x', y', z')
- For each visible surface point in eye space transform to *light space*
  - $(x_c, y_c, z_c) \Rightarrow (x_l, y_l, z_l)$
- If $z_l > z'$ then point is in shadow

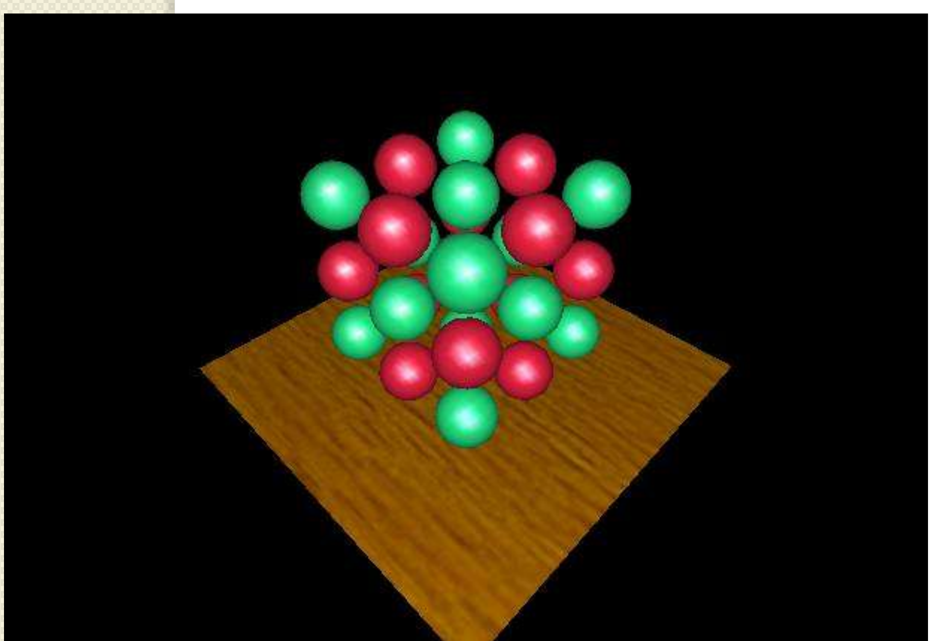
Shadow Map

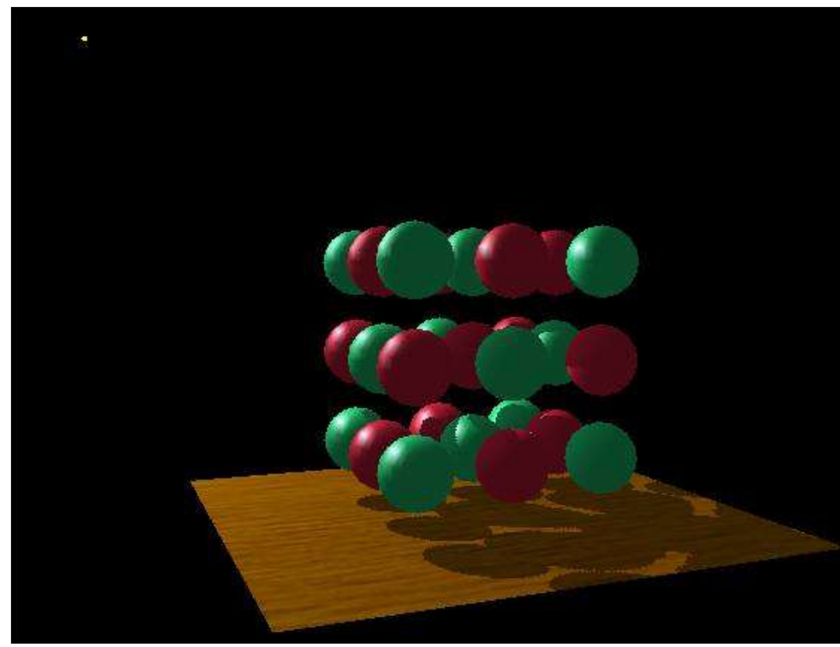
- Visible surface point E is in shadow and occluded by point L when transformed to *light space*

If L is closer to the light than E, then E is in shadow



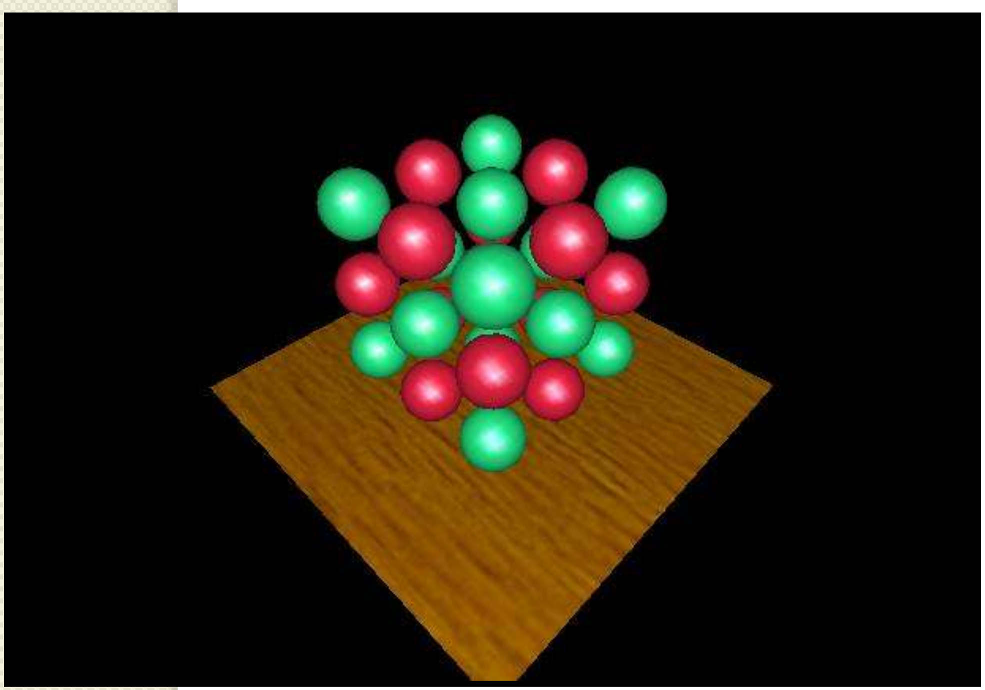
Shadow Map : Two Pass Approach

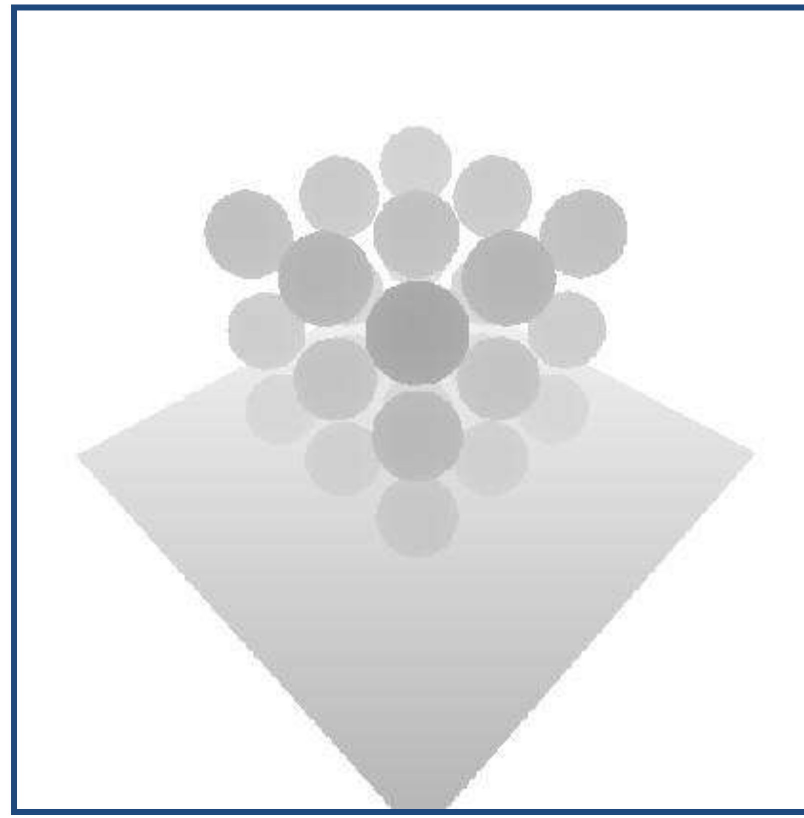




1st Pass

View from light

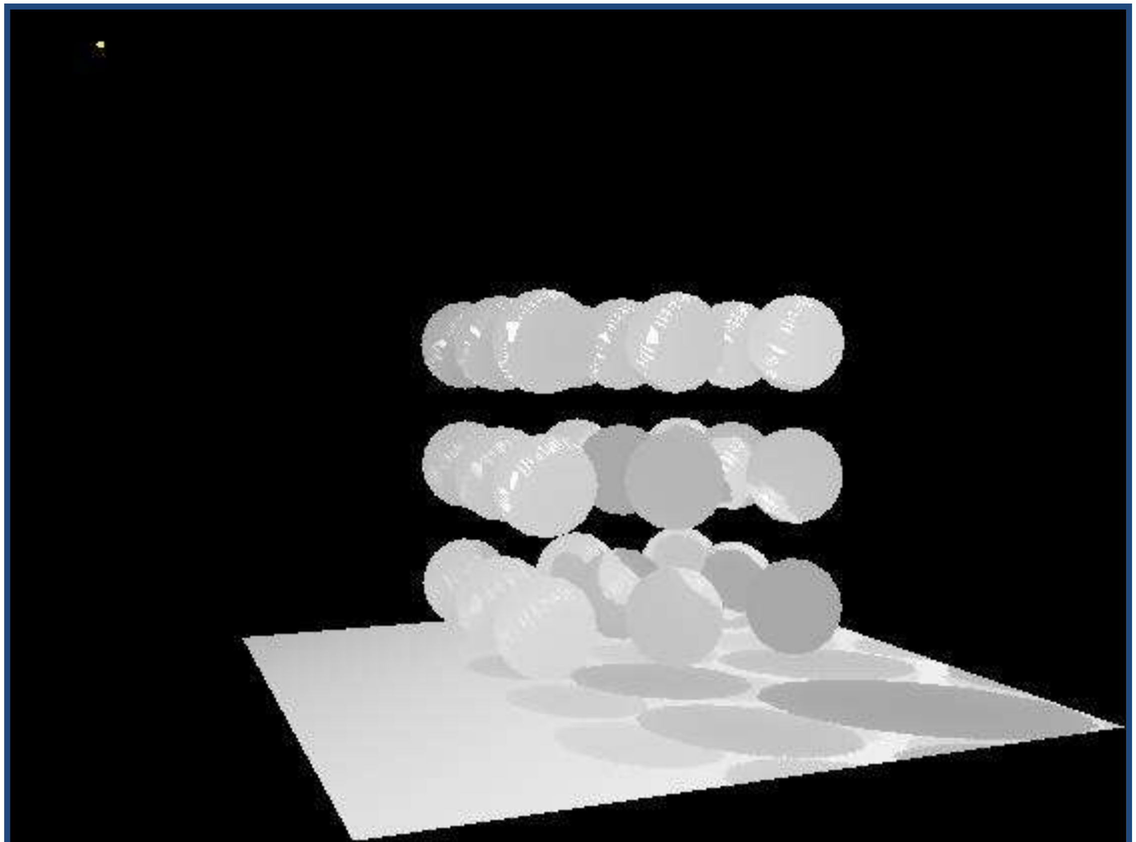




Depth Buffer

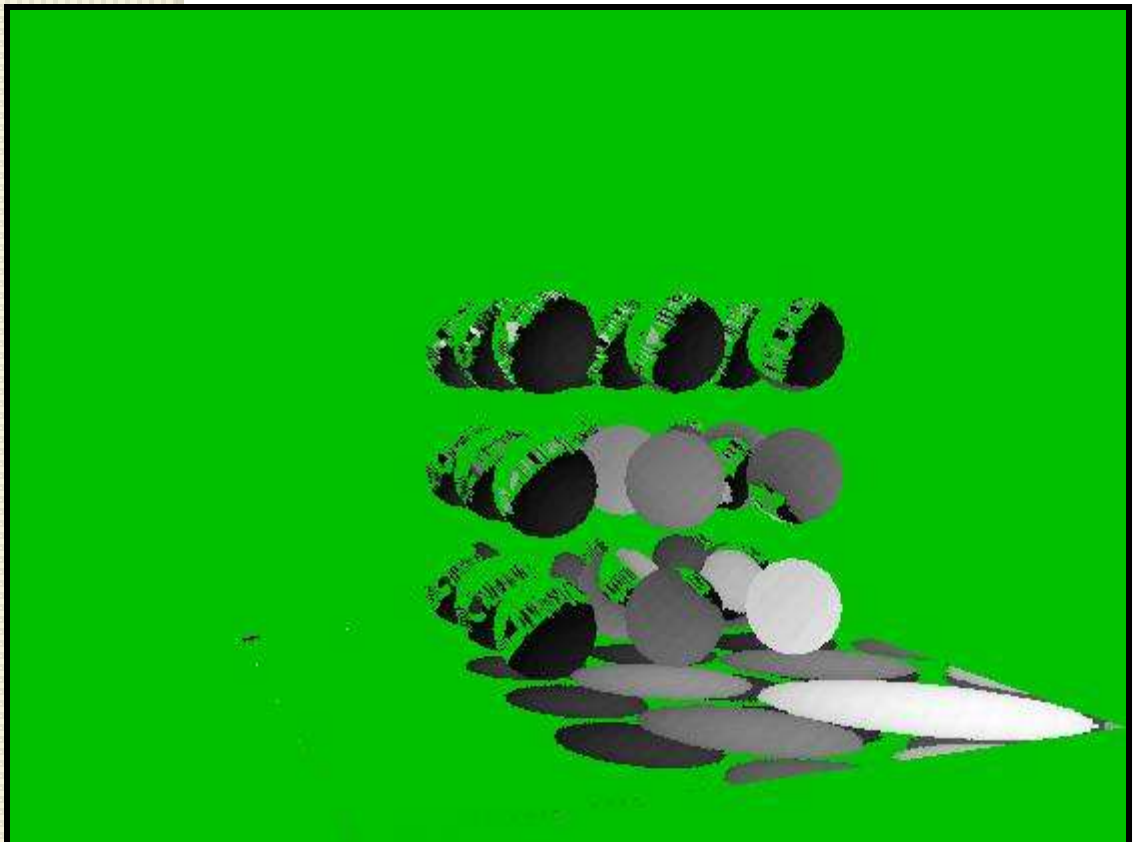
2nd Pass

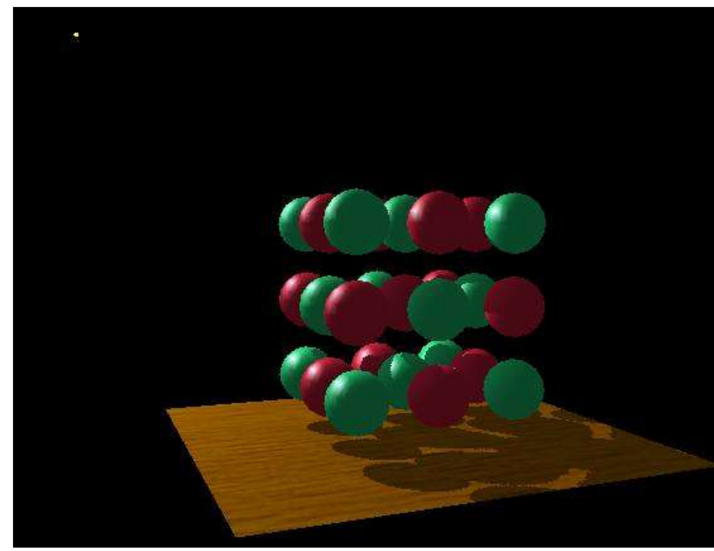
Visible surface depth



2nd Pass

Non-green in shadow





Final Image

Shadow Maps With Graphics Hardware

- Render scene using the light as a camera 
- Read depth buffer out and copy to a 2D texture.
  - Rather than Binary projected shadow, we now have a depth texture.

- Fragment's light position can be generated using eye-linear texture coordinate generation
  - specifically OpenGL's GL_EYE_LINEAR texgen
  - generate homogenous (s, t, r, q) texture coordinates as light-space (x, y, z, w)

The Shadow Mapping Concept (1)

- Depth testing from the light's point-of-view
  - Two pass algorithm
  - First, render depth buffer from the light's point-of-view
    - the result is a "depth map" or "shadow map"
    - essentially a 2D function indicating the depth of the closest pixels to the light
  - This depth map is used in the second pass

The Shadow Mapping Concept (2)

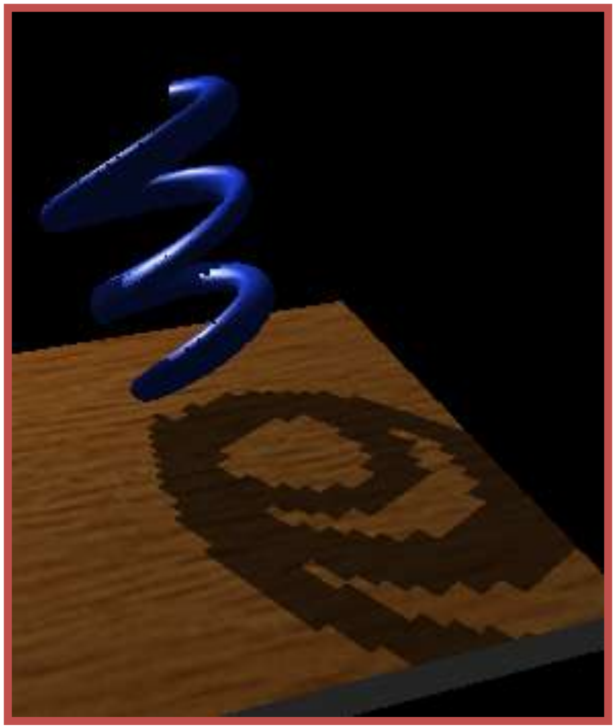
- Shadow determination with the depth map
  - Second, render scene from the eye's point-of-view
  - For each rasterized fragment
    - determine fragment's XYZ position relative to the light
    - this light position should be setup to match the frustum used to create the depth map
    - compare the depth value at light position XY in the depth map to fragment's light position Z

The Shadow Mapping Concept (3)

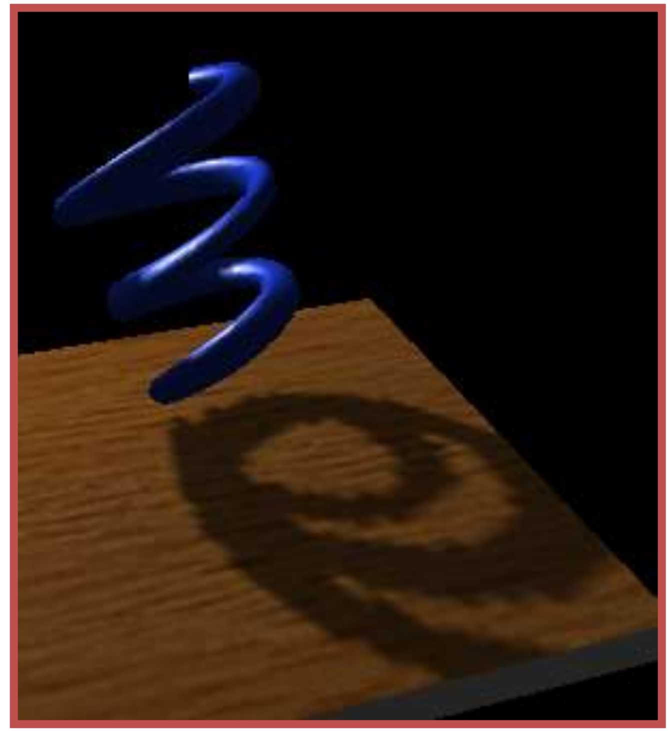
- The Shadow Map Comparison
  - Two values
    - A = Z value from depth map at fragment's light XY position
    - B = Z value of fragment's XYZ light position
  - If B is greater than A, then there must be something closer to the light than the fragment
    - then the fragment is shadowed
  - If A and B are approximately equal, the fragment is lit

Hardware Shadow Map Filtering Example

GL_NEAREST: blocky



GL_LINEAR: antialiased edges



Low shadow map resolution
used to heighten filtering artifacts

Issues with Shadow Mapping

- Not without its problems
  - Prone to aliasing artifacts
    - “percentage closer” filtering helps this
    - normal color filtering does **<u>not</u>** work well
  - Depth bias is not completely foolproof
  - Requires extra shadow map rendering pass and texture loading
  - Higher resolution shadow map reduces blockiness
    - but also increases texture copying expense

Issues with Shadow Mapping

- Not without its problems
  - Shadows are limited to view frustums
    - could use six view frustums for omni-directional light
  - Objects outside or crossing the near and far clip planes are not properly accounted for by shadowing
    - move near plane in as close as possible
    - but too close throws away valuable depth map precision when using a projective frustum