Chapter 8

Three-Dimensional Object Representations

8.1 Overview

The main goal of three-dimensional computer graphics is to generate two-dimensional images of a scene or of an object based on a a description or a model.

The internal representation of an object depends on several implications:

- The object may be a real object or it exists only as a computer representation
- The manufacturing of the object is bound closely to the visualization:
  - Interactive CAD systems
  - Modeling and visualization as a tool during design and manufacturing
  - More than just 2-D output possible!

8.1 Overview

Implications (continued)

- The precision of the internal computer representation depends on the application. For example, an exact description of the geometry and shape in CAD applications vs. an approximation sufficient for rendering of the object.
- For interactive applications, the object may be described by several internal representations. These representations may be generated in advance or on-the-fly.
  - Level-of-detail (LOD) techniques

8.1 Overview

The modeling and representation of an object involves the following in particular:

- Generation of 3-D geometry data

  CAD interface, digitizer, laser scanner (reverse engineering), analytic techniques (e.g. sweeping), image (2-D) and video (3-D) analysis

- Representation, efficient data access and conversion

  Polygonal nets (e.g. triangulation), is the most common representation for rendering objects. Alternatives: finite elements (FEM), constructive solid geometry (CSG), boundary representation (B-rep), implicit surfaces (isosurfaces), surface elements (surfels = points + normals), …

- Manipulation of objects (change shape, …)

  e.g. Boolean operations, local smoothing, interpolation of features (e.g. boundary curves), “engraving” of geometric details, …

8.1 Overview

The topics of this chapter will be:

- Polygonal representations
- Rendering Polygons with OpenGL
- Quadric surfaces
- Blobby Objects
- Octree, BSP tree

8.2 Polygonal Representation

Properties/Characteristics:

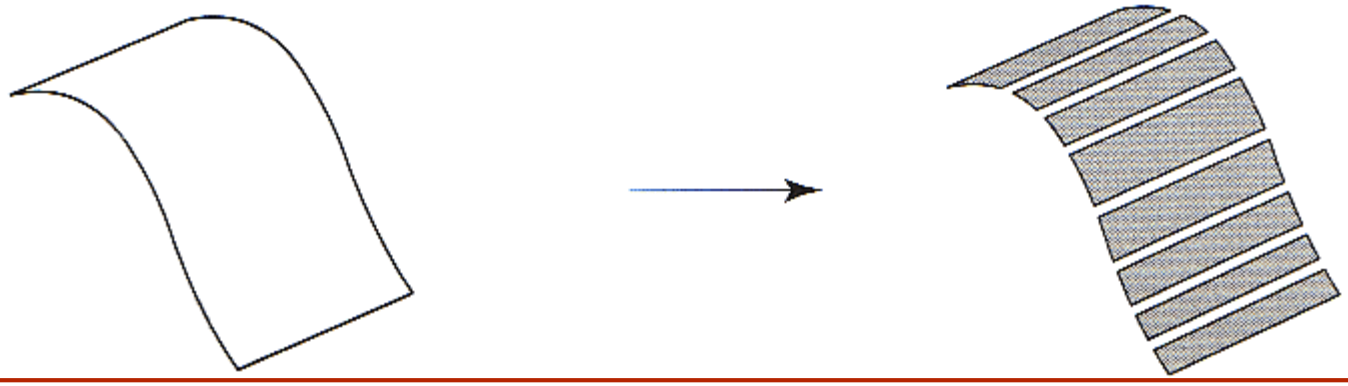
- The precision of the approximation (number and size of polygons) can be chosen depending on the application, but several questions arise, e.g.:
  - What polygonal resolution is required for a precise representation?
  - What polygonal resolution is required for the renderer to make the piecewise approximation appear smooth?
  - What is the correlation between number of polygons and the size of the final display of the object?

  ➞ Often the following rule of thumb is used: Choose the polygonal resolution based on the curvature of the object

8.2 Polygonal Representation

Properties/Characteristics:

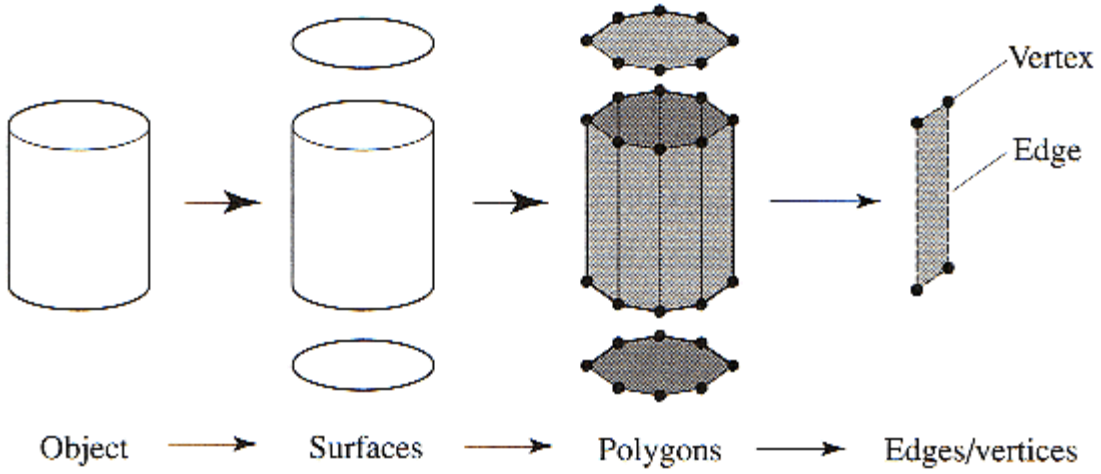
- Classic representations of three-dimensional objects in computer graphics
- Object is represented by a net of polygonal surfaces (usually triangles) ➞ piecewise linear interpolation
- The polygonal surfaces are usually an approximation of the curved surface, representing the object's boundary.



8.2 Polygonal Representation

Hierarchy of the representation:

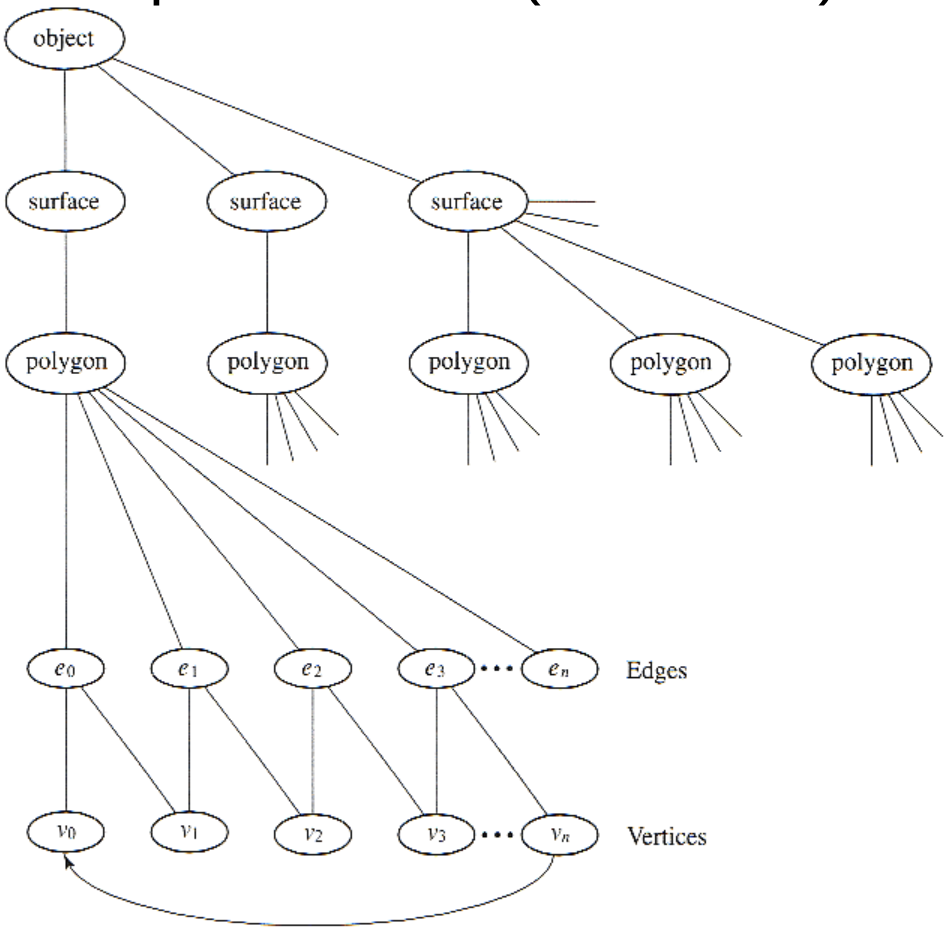
Concept: The object constitutes of several surface elements. Each surface element is represented by several polygons. Every polygon has vertices and edges.



8.2 Polygonal Representation

Hierarchy of the representation (continued):

Topology:



object

surfaces

polygons

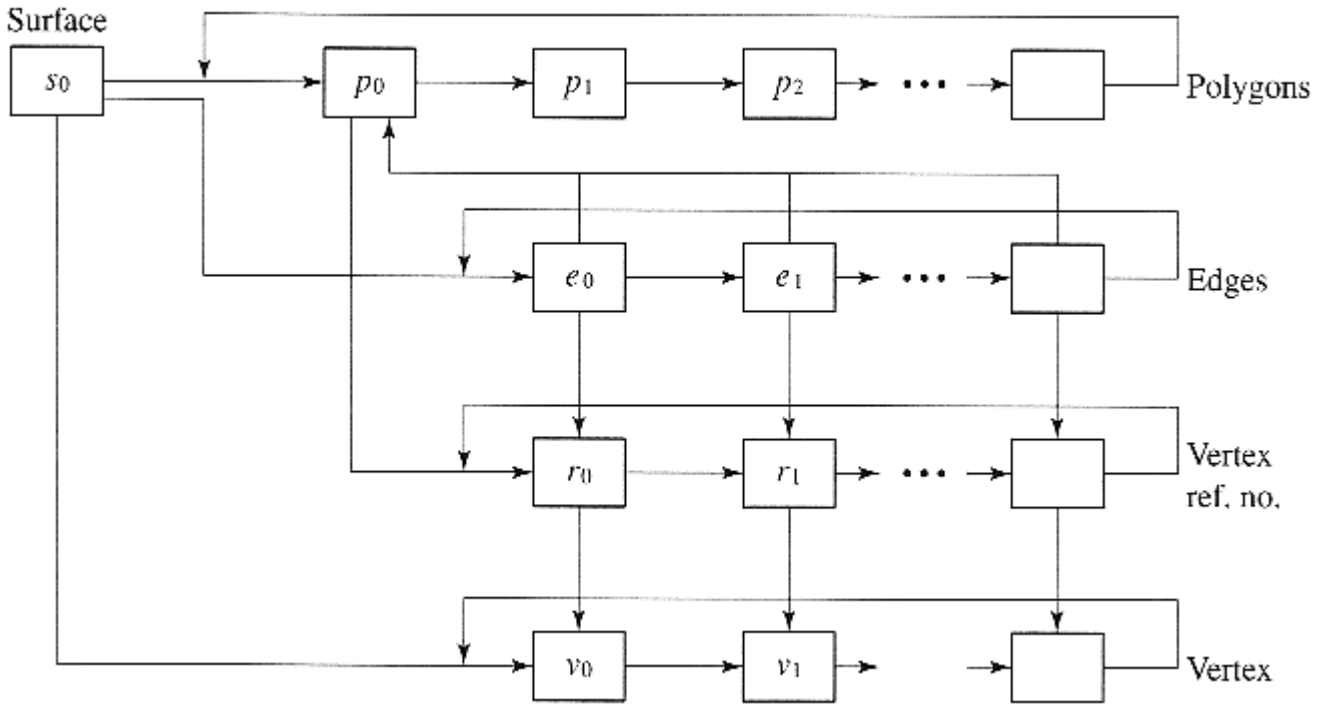
edges

vertices

8.2 Polygonal Representation

Hierarchy of the representation (continued):

Data structure:



Vertices are stored only once

8.2 Polygonal Representation

Comment on data structures:

Data structures can contain – besides geometry information – special attributes required for the application or for the rendering:

- Surface attributes:

  Representation (triangle, polygon, free-form surface), coefficients, normal vector, properties (plane, convex, holes, …), reference to vertices (and edges, if necessary)

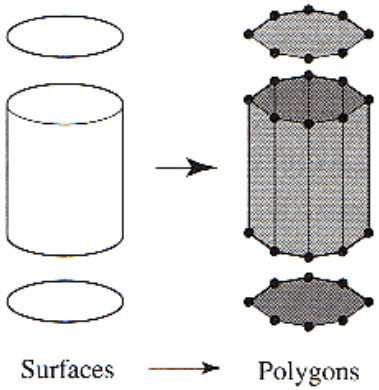
- Edge attributes:

  Length, type (round edge, feature line, virtual edge, reference to vertices and/or polygon

- Vertex attributes:

  Normal vector, color, texture coordinates, reference to polygon and/or edge

8.2 Polygonal Representation



Comment on edges:

Obviously, there are two different kinds of edges involved in the approximate representation:

- Sharp edges (feature lines)
  - This type of edge should be visible
- Virtual edges ("inside" a smooth surface)
  - These should be invisible after rendering
  - Interpolative shading algorithms

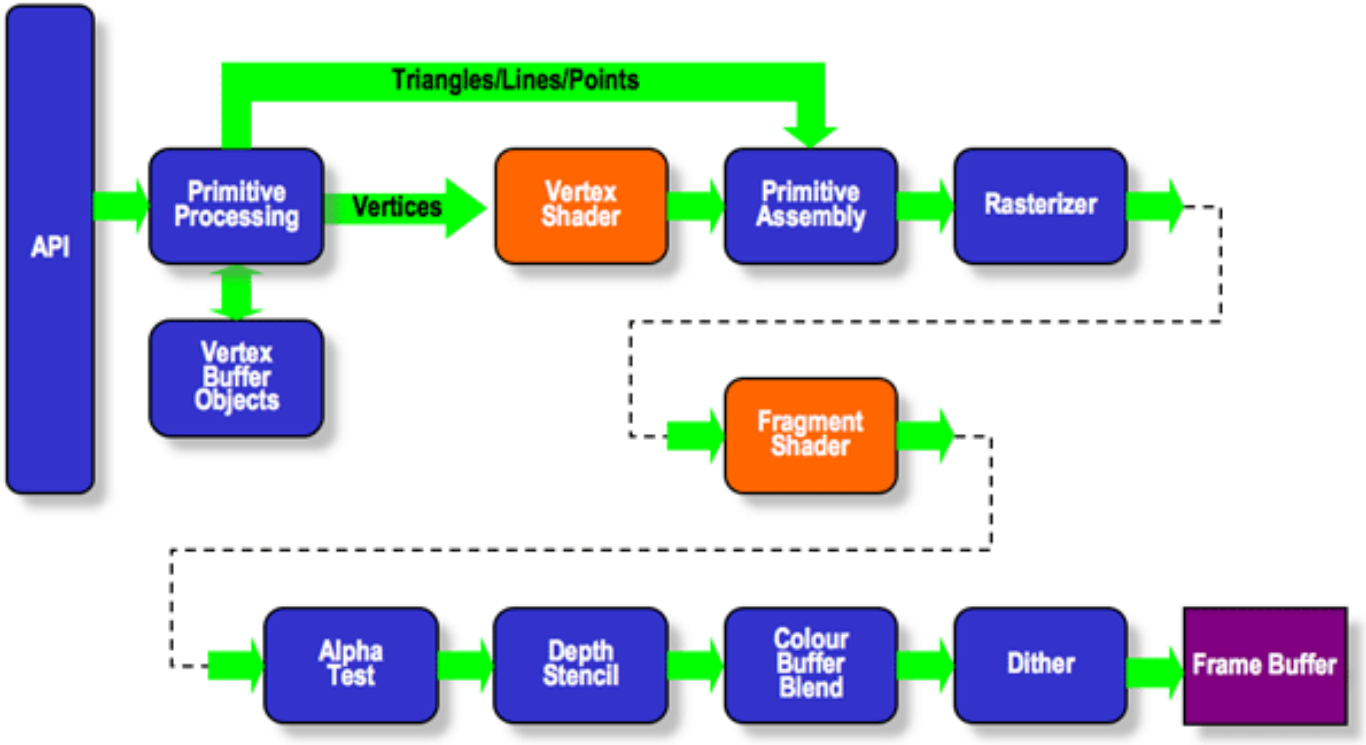
  ➞ flat, Gouraud, Phong shading (now implemented in hardware)

Which kind of edge is to be used can be enforced by the data structure by storing edges multiple times (see image).

8.3 Polygon Rendering with OpenGL

OpenGL rendering pipeline:

Both, vertex and fragment shader are programmable



8.3 Polygon Rendering with OpenGL

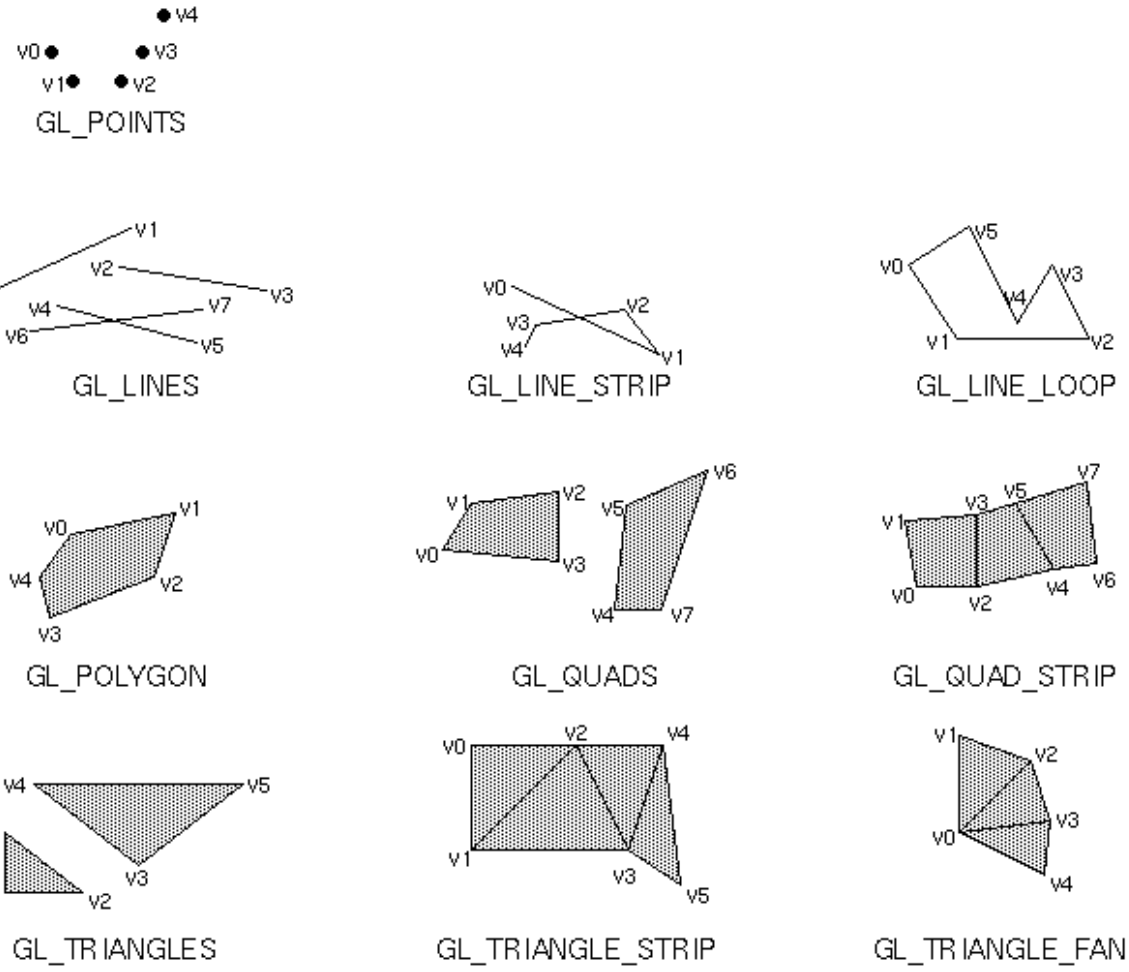
OpenGL supports several types of polygons:

```
GL_POLYGON
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS
GL_QUAD_STRIP
```

Convenience functions exist for certain objects:

```
glutSolidTetrahedron glutWiredTetrahedron
glutSolidCube        glutWireCube
...
```

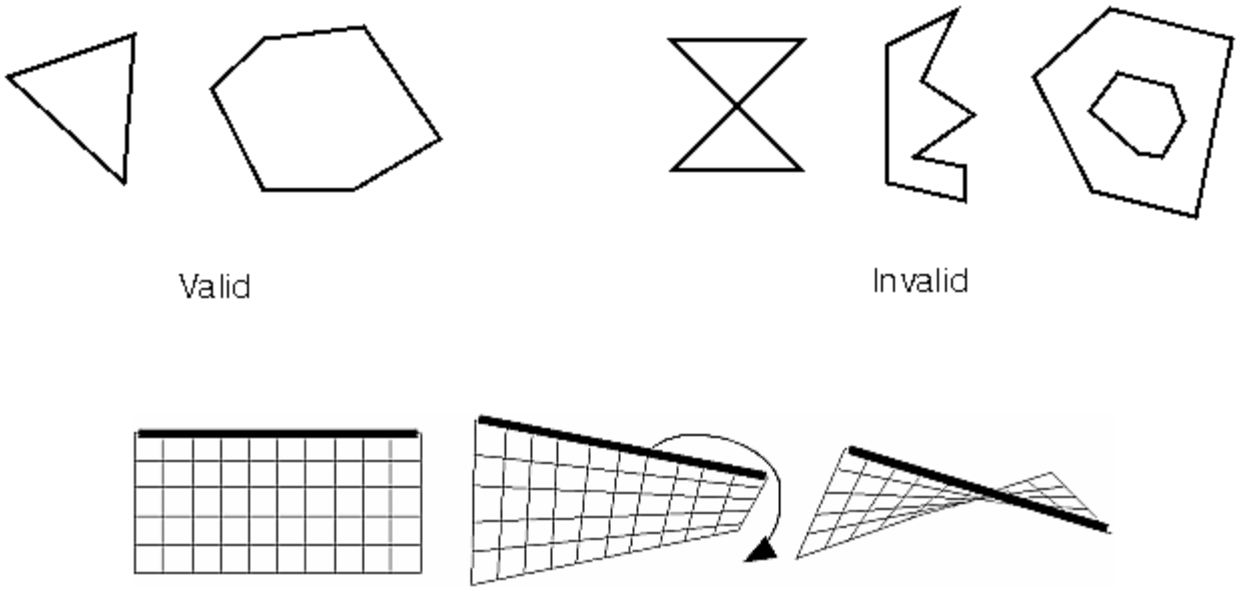
8.3 Polygon Rendering with OpenGL



8.3 Polygon Rendering with OpenGL

Beware:

OpenGL will ignore invalid polygons, e.g. self intersecting, non-convex, or non-planar polygons



8.3 Polygon Rendering with OpenGL

There are basically four different ways to render geometric objects with OpenGL:

- Direct rendering
- Display lists
- Vertex arrays
- Vertex buffer objects
- Vertex array objects

8.3 Polygon Rendering with OpenGL

Direct rendering:

```
glBegin (GL_TRIANGLES);
glNormal3f ( … );
glVertex3f ( … );
…
glNormal3f ( … );
glVertex3f ( … );
glEnd ();
```

In case of polygons with a fixed number of vertices, i.e. triangles, quads, etc., you can generate several such polygons using one `glBegin`/`glEnd` block.

8.3 Polygon Rendering with OpenGL

Display lists:

Stores OpenGL API commands in graphics memory for faster access.

```
GLuint index = glGenLists (1);
if (index != 0) {
  glNewList (index, GL_COMPILE);
   … // draw something
  glEndList ();
}
glCallList (index);
```

Using `GL_COMPILE_AND_EXECUTE` instead of `GL_COMPILE` makes the `glCallList` unnecessary.

8.3 Polygon Rendering with OpenGL

Vertex arrays:

Store vertices in bulk arrays to reduce number of OpenGL function calls.

```
GLfloat vertices[] = { … };
GLfloat normals[] = {… };
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_NORMAL_ARRAY);
glNormalPointer (GL_FLOAT, 0, normals);
glVertexPointer (3, GL_FLOAT, 0, vertices);
glDrawArrays (GL_TRIANGLE_STRIP, 0, 10);
```

This constructs a triangle strips using the first ten elements. The 0 as argument for the arrays is the stride parameter allowing you to skip elements within the arrays.

8.3 Polygon Rendering with OpenGL

Vertex buffer objects (VBO):

Vertex buffer objects are like vertex arrays, but stored in graphics memory for faster access.

Fill the VBO with data; use indices to remember them:

```
GLuint vbovertices, vbonormals;
GLfloat vertices[] = { … }, normals[] = {… };
glGenBuffers (1, vbovertices);
glGenBuffers (1, vbonormals);
glBindBuffer (GL_ARRAY_BUFFER vbovertices);
glBufferData (GL_ARRAY_BUFFER, datasize,
              vertices, GL_STREAM_DRAW);
glBindBuffer (GL_ARRAY_BUFFER, vbonormals);
glBufferData (GL_ARRAY_BUFFER, datasize,
              normals, GL_STREAM_DRAW);
```

8.3 Polygon Rendering with OpenGL

Vertex buffer objects (continued):

Now, draw the previously generated VBOs:

```
glBindBuffer (GL_ARRAY_BUFFER vbovertices);
glVertexPointer (3, GL_FLOAT, 0, (GLvoid *)0);
glBindBuffer (GL_ARRAY_BUFFER, vbonormals);
glNormalPointer (GL_FLOAT, 0, (GLvoid *)0);
glDrawArrays (GL_TRIANGLE_STRIP, 0, count);
```

Notes:

- There is no actual data pointer required for the `glVertexPointer` and `glNormalPointer` calls since the VBOs are used as data repository.
- The client states need to be set just like with vertex arrays

8.3 Polygon Rendering with OpenGL

Vertex Array Objects (VAOs)

- VAOs can store the data of a geometric object to reflect its state so that the OpenGL driver can use this information for optimization
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an object
  - previously, you might have needed to make many calls to make all the data current

8.3 Polygon Rendering with OpenGL

Implementing VAOs

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

8.3 Polygon Rendering with OpenGL

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO, so that the VAO can reference to the VBO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`
  - bind VAO for use in rendering `glBindVertexArray()`

8.3 Polygon Rendering with OpenGL

Implementing VAOs

Create buffer indices/"names":

```
GLuint buffers[2];

glGenBuffers(1, &buffers);

glGenBuffers(ARRAY_SIZE_IN_ELEMENTS(buffers),
             buffers);
```

8.3 Polygon Rendering with OpenGL

Creating VBOs for the VAOs

Uploading the vertex data to the graphics card:

```
glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(GL_ARRAY_BUFFER, 0, sizeof(points),
             points, GL_STATIC_DRAW);
```

We now need to tell OpenGL what structure our arrays have using glVertexAttribPointer:

```
void glVertexAttribPointer(index, size, type,
                           normalized, stride, pointer);
```

Following our example, this could look like this:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

8.3 Polygon Rendering with OpenGL

Creating VBOs for the VAOs

Uploading the color data to the graphics card:

```
glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(points),
             sizeof(colors), colors, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

Note that this creates a **structure of arrays** (SOA) that the VAO handles itself without us having to consolidate all data in a single **array of structures** (AOS).

8.3 Polygon Rendering with OpenGL

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays(GL_TRIANGLES, 0,
                NumVertices);
```

- Usually invoked in display callback
- You can use something like

```
glBindVertexArray(0);
```

to avoid overwriting your array (you will need to switch back for rendering).

8.4 Quadric Surfaces

Quadric surfaces are described with second-degree equations (quadrics). Quadratic surfaces are common elements in computer graphics and CAD. Some examples are:

Sphere: $$x^2 + y^2 + z^2 = r$$

Ellipsoid: $$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

8.4 Quadric Surfaces

OpenGL supports quadric surfaces directly using the GLUT or GLU libraries.

For example:

```
glutSolidSphere (r, xdiscretization,
                 ydiscretization);
```

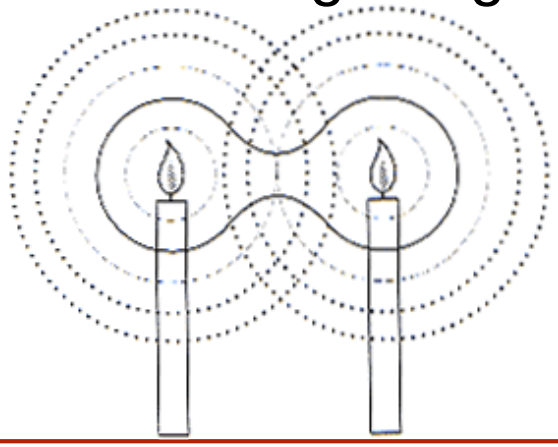
Or:

```
GLUQuadric *quadric;
quadric = gluNewQuadric ();
gluSphere (quadric, r, xdiscretization,
           ydiscretization);
```

8.5 Blobby Objects

Idea: describe the surface or volume of an object as iso-surface within a scalar field (i.e. a point is part of the iso-surface if and only if the scalar field has the same so-called iso-value). The scalar field itself is generated through generating primitives (functions).

Example: point heat sources create a spherical field. By adding two of those fields we get a global scalar field.

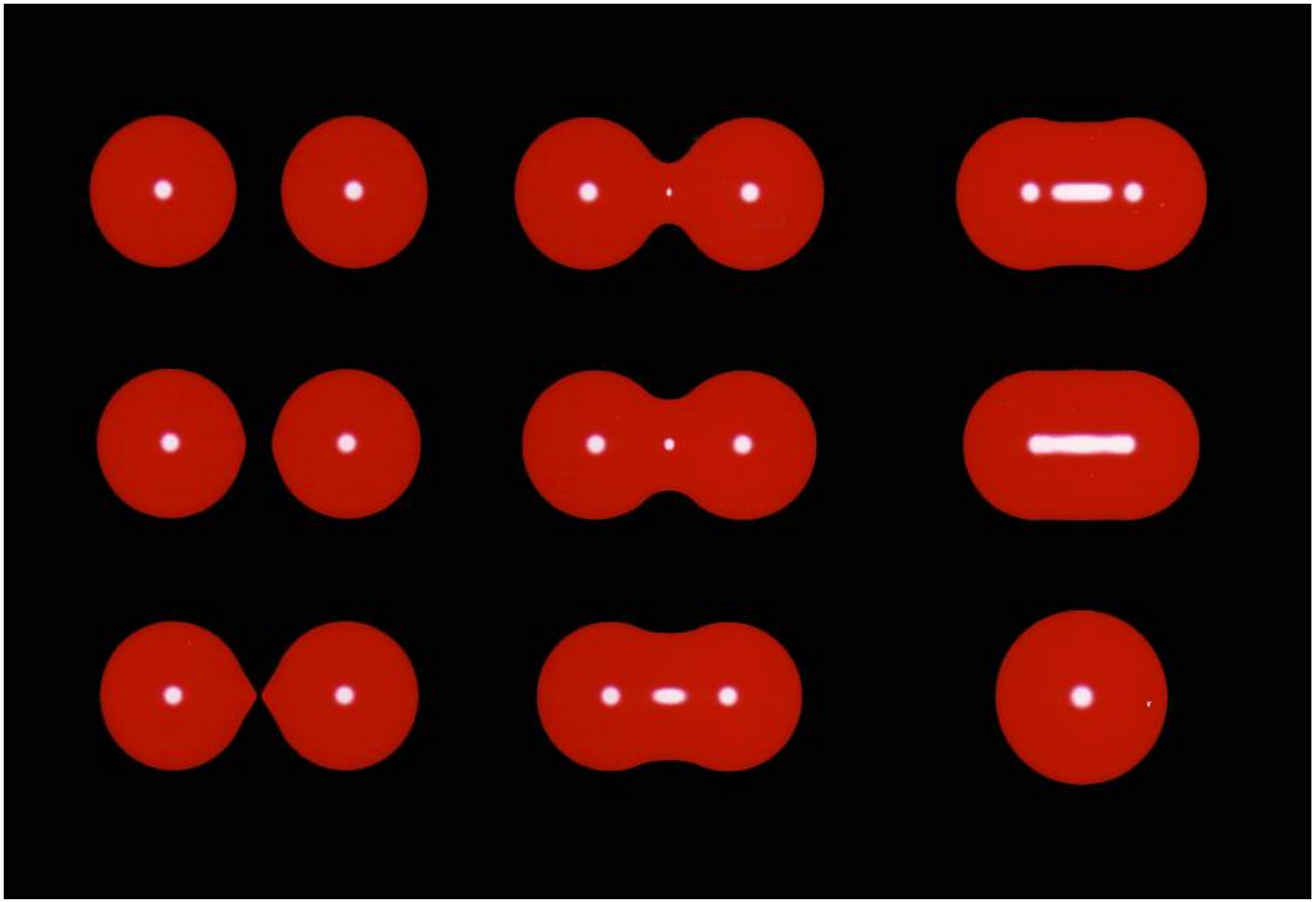


8.5 Blobby Objects

The appearance of the iso-surface is relatively easy to handle if the center points and the individual scalar fields are chosen reasonably. The following image shows two iso-surfaces which are generated by two radial symmetric fields. The two centers of the generating fields approach each other when going from top to bottom and left to right in the image until they are at the exact same location.

You can see the merging effect during the transition of the smooth iso-surface (C^1-continuous in this case) after the two centers get close enough. In the opposite case, where the centers move apart we would see the iso-surface separating.

8.5 Blobby Objects



8.5 Blobby Objects

Density function:

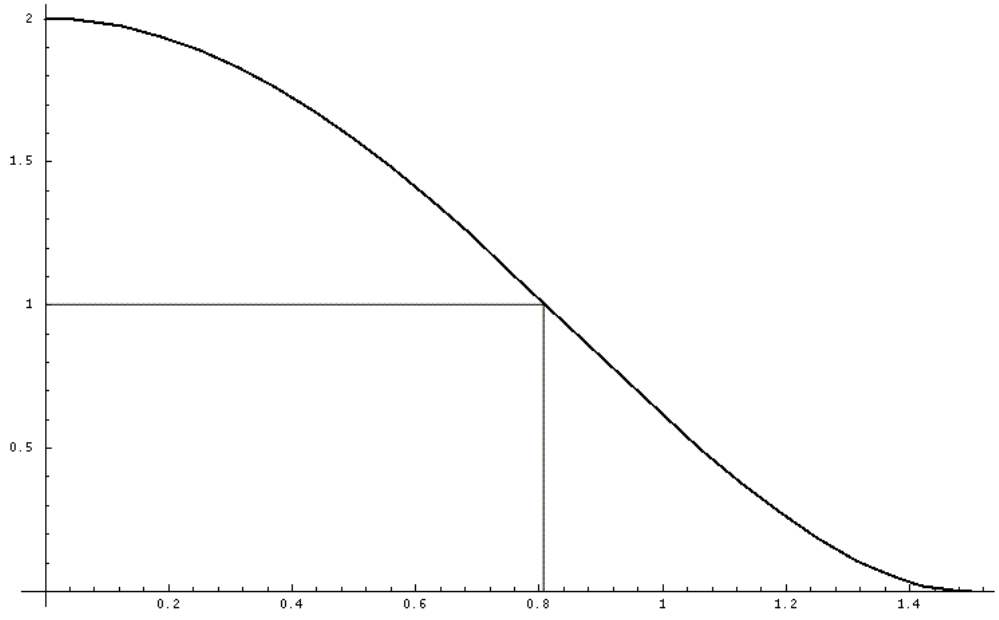
The most commonly used density functions $f : \Re_0^+ \rightarrow [0, a]$ with $a \in \Re, a > 0$ have the following properties:

- For $t \in [0, b] (b \in \Re, b > 0)$ f is a polynomial
- $f(t) = 0$ for $t > b$ is the maximal radius
- $f(0) = a, f(b) = 0$
- $f'(0) = 0, f'(b) = 0$
- f is monotonically decreasing

The following graph shows an example for such a density function.

8.5 Blobby Objects

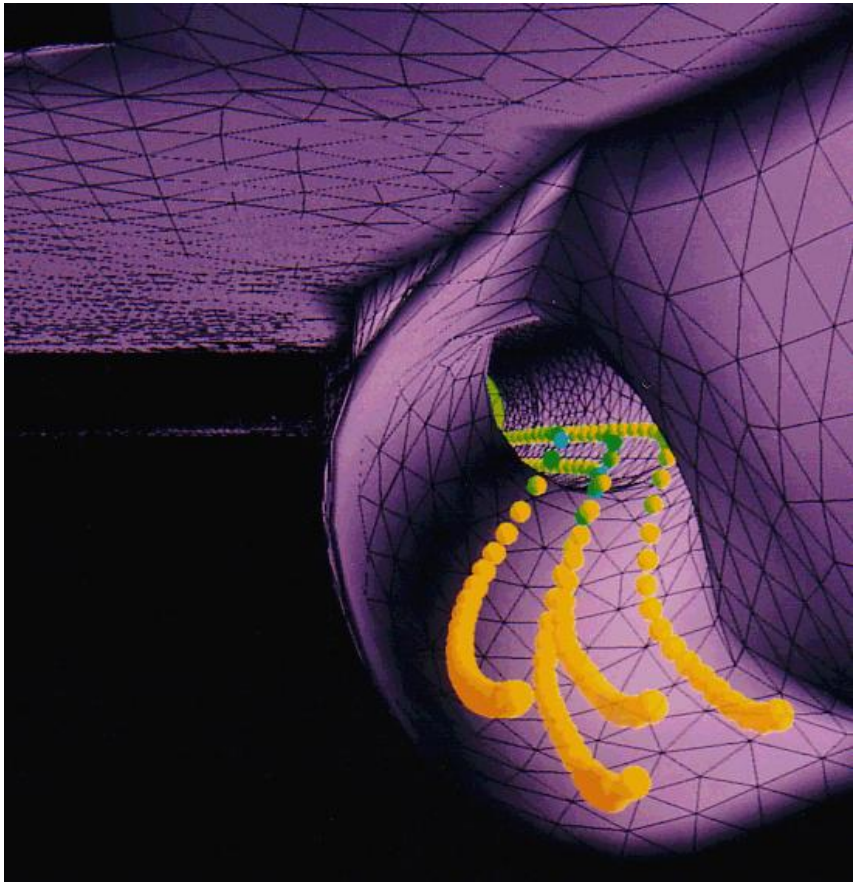
Density function (continued):



Then, a radial symmetric field of a discrete blob at a point *P* can be defined as $F_d : \Re^3 \rightarrow \Re$ with $F_d(x) = f(\|x - P\|_2)$

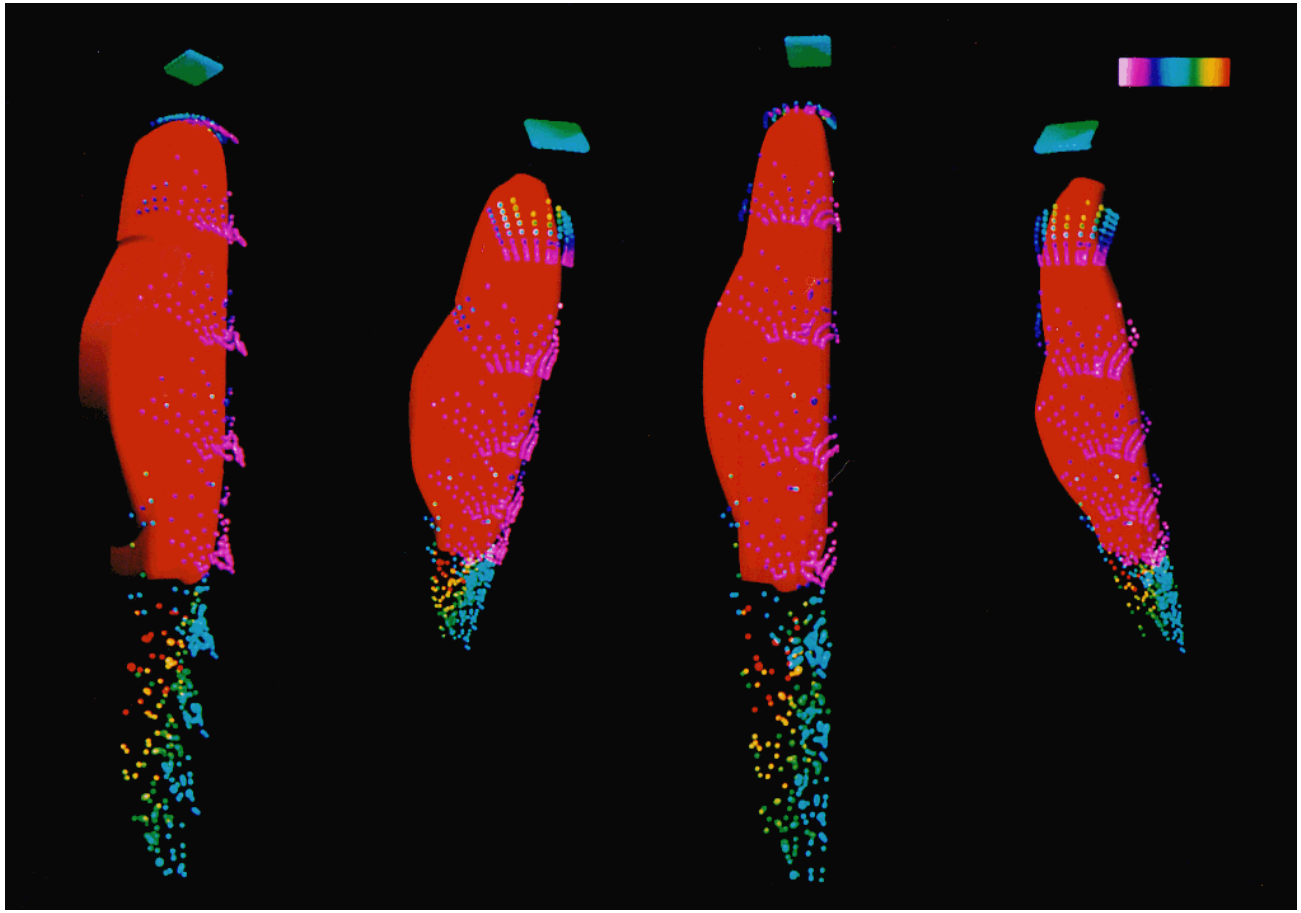
8.5 Blobby Objects

Application example: flow simulation



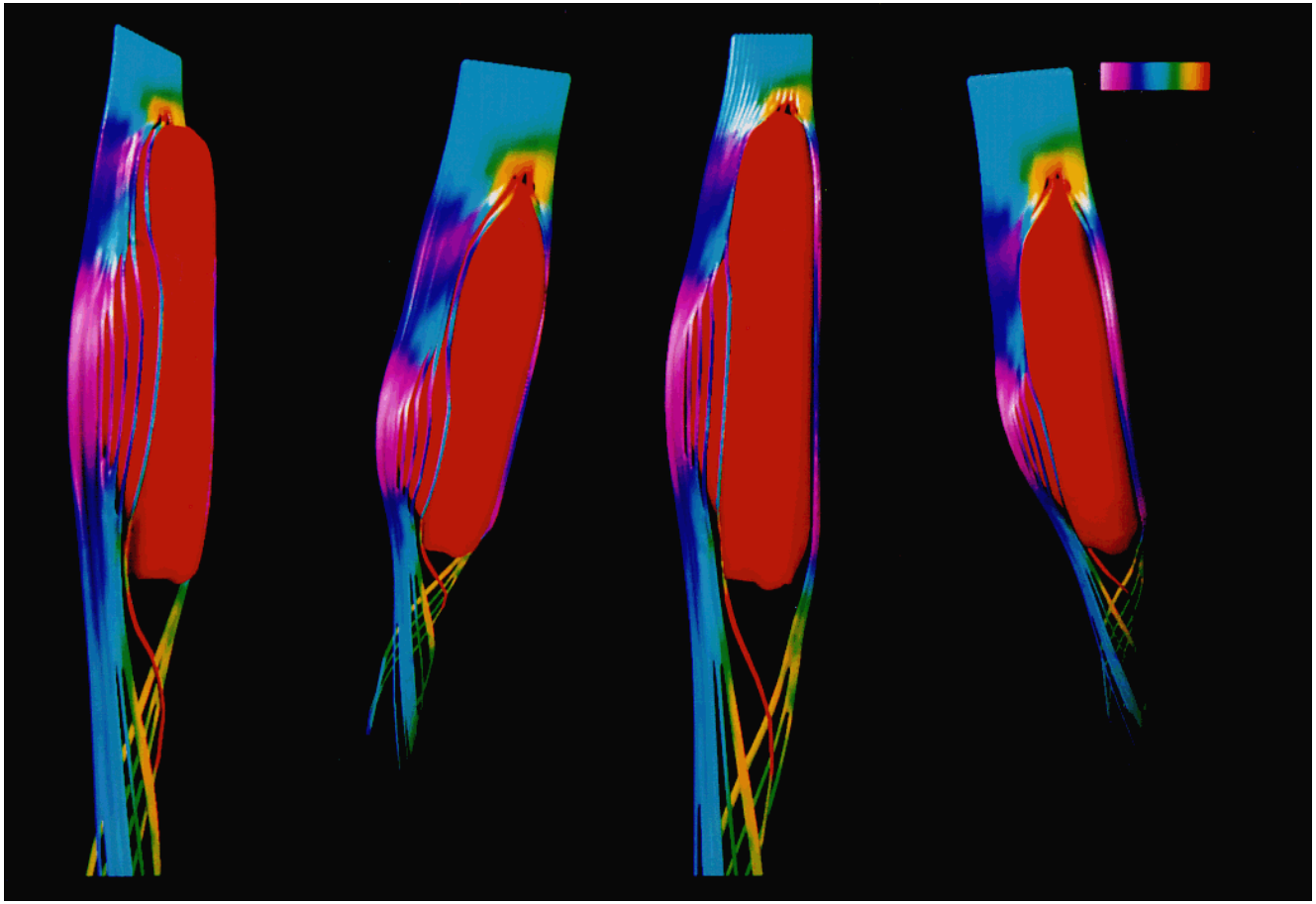
8.5 Blobby Objects

Application example: flow simulation



8.5 Blobby Objects

Application example: flow simulation



8.8 Space Partitioning

For representing an object using space subdivision techniques the object space is split up into several smaller elements. For each element, we store if this specific element is covered by the object.

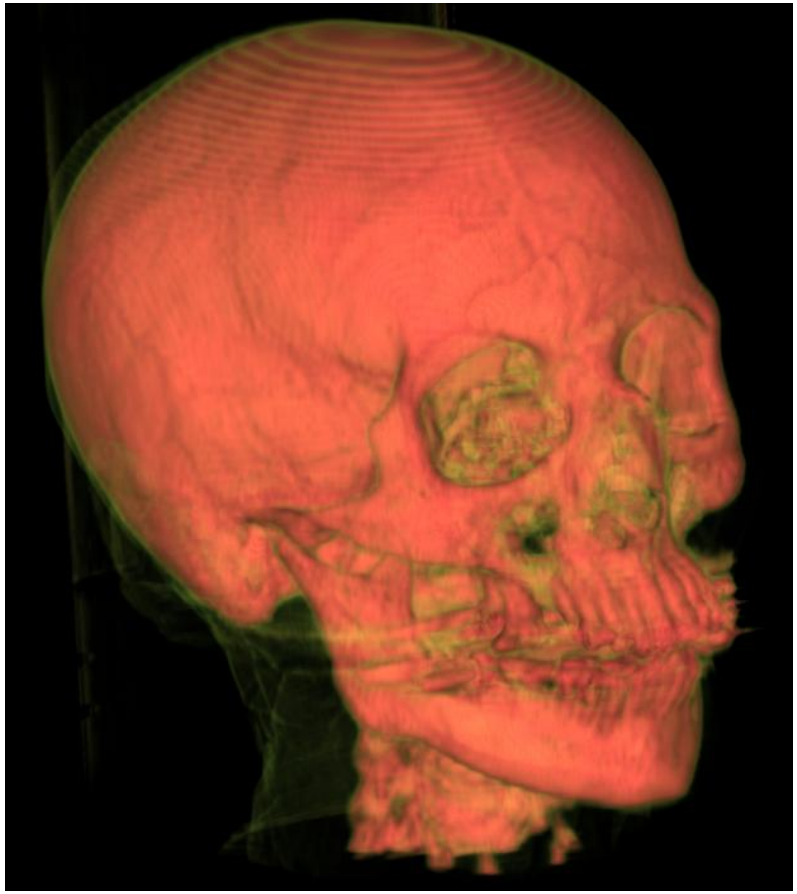
Standard approach:

- Space is divided by a regular equidistant grid, resulting in a grid where each cell has exact identical geometry.
- In 3-D space, we get cube-shaped cells, which are called **voxels** (volume element).

  → The name is in analogy to pixel (picture element) in 2-D

8.8 Space Partitioning

Example: volumetric image of a CT-scanned object



8.8 Space Partitioning

Advantages:

- It can be determined very easily if a given point is part of the object or not.
- It can be checked easily if two objects are connected or attached to each other.
- The representation of an object is unique.

Disadvantages:

- There cannot be any cells that are only partly filled.
- Objects can generally be represented approximately.
- For a resolution of n voxels in each dimension we need n^3 voxels to represent the object. Therefore, it requires a lot of memory → save space using octrees.

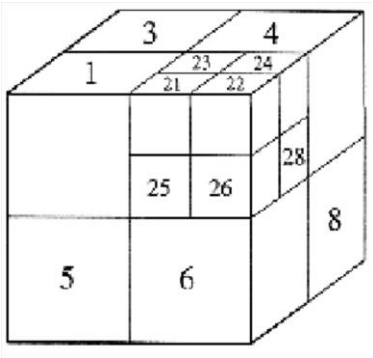
8.8 Space Partitioning

Octrees

An octree is a hierarchical data structure for storing an irregular, i.e. not equidistant, sub-division of a 3-D space.

Idea:

- The initial element is a cube which covers the entire object space. The element can have two states: covered or uncovered.
- In case an element is partly covered, it is sub-divided into eight equally sized sub-elements.
- The coverage of each element is checked recursively until a desired resolution is achieved.



8.8 Space Partitioning

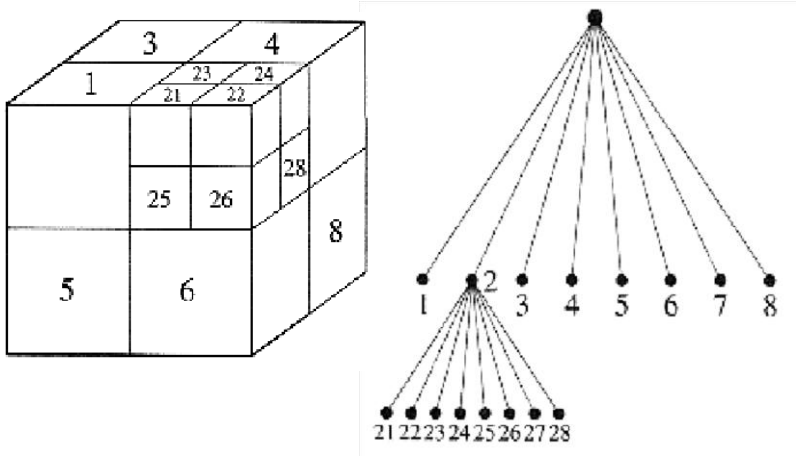
Octrees (continued)

In an octree, each node (element) that is not a leaf has eight successors (sub-elements).

The root of the tree represents the initial cube. For each sub-division a fixed numbering scheme is used for the sub-elements when inserting a new node as a child.

Each leaf stores the state of its corresponding (sub-) cube.

Each inner node represents a partly covered cube.



8.8 Space Partitioning

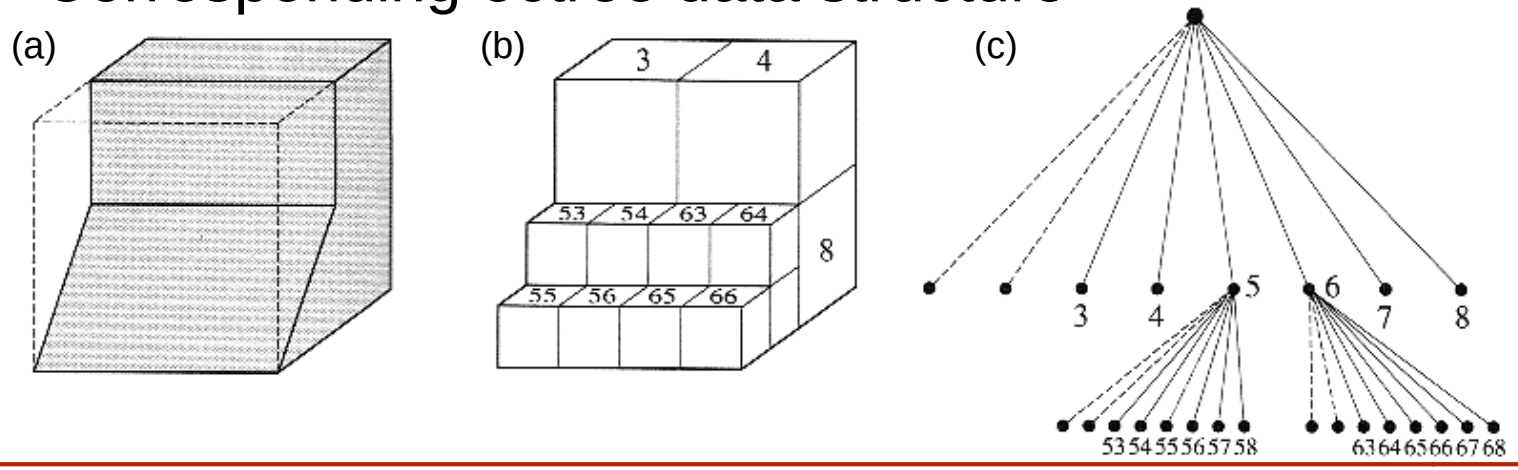
Octrees (continued)

Example: representation of a 3-D object using an octree

a) Object embedded into initial cube.

b) Representation of the object using a maximal sub-division of two.

c) Corresponding octree data structure



8.8 Space Partitioning

Octrees (continued)

Octrees can not only be used for representing 3-D objects. A very common use of octrees is the sub-division of a 3-D scene.

- Here, the individual objects are represented by standard data structures, e.g. polygons.
- The state of the cells of the octree is then extended to a data structure that stores a list of objects, e.g. polygons, which are contained by the cell.

This results in a significant performance increase for algorithms that work on the individual areas of the object space locally (e.g. ray tracing).

8.8 Space Partitioning

Quadtrees

The principle of sub-dividing the 3-D space can be generalized to an n-dimensional space.

For the case $n=2$ we get the sub-division of a 2-D plane resulting in a quadtree, where each inner node of the tree has exactly four children.

Historically, quadtrees are the older data structures. They were used initially in the late 60's of the last century. Octrees were derived from quadtrees and used since the late 70's, early 80's of the last century.

8.8 Space Partitioning

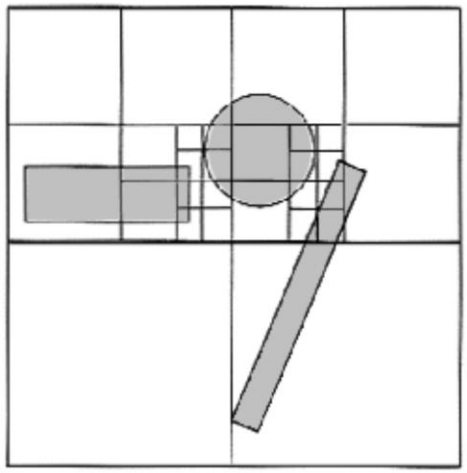
Quadtrees (continued)

Example: sub-division of a 2-D space using a quadtree.

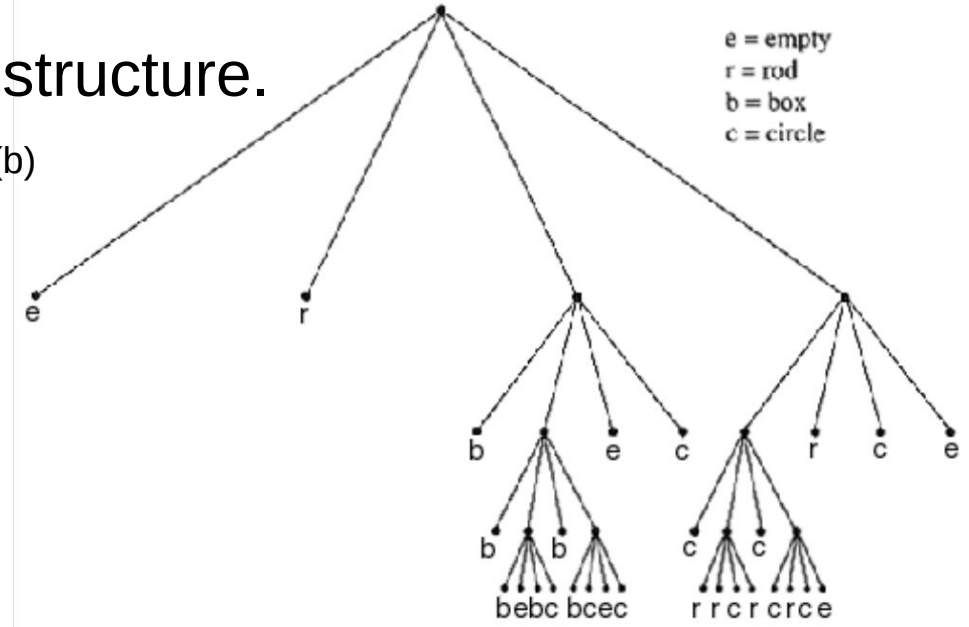
a) Sub-division of the 2-D space until each cell containes maximally one object.

b) Corresponding data structure.

(a)



(b)



8.8 Space Partitioning

Binary space-partitioning (BSP trees)

Octrees and quadtrees both sub-divide at each level equally in each dimension, i.e. at the center.

A BSP tree offers an alternative representation where an element can be sub-divided into **two** sub-elements at an arbitrary (hyper-)plane

- If one sub-element is defined as part of the inside while the other sub-element is defined as the outside, a convex polyhedron can be represented by using properly chosen planes limiting the volume.
- By uniting convex interior areas, arbitrary concave polyhedra with holes can be defined.

8.8 Space Partitioning

BSP trees (continued)

In the realm of computer graphics, BSP trees are often used for determining the visibility of an object.

Idea:

- BSP trees can – similar to octrees and quadtrees – be used for sub-dividing a 3-D scene (see next example). Here, the objects are not bound to a particular rasterization.
- The object space is to be sub-divided recursively in such a way, that each area contains at most one object.
- Using the locations of those areas relatively to the view point, the objects can be sorted according to the viewing distance (depth) easily, i.e. it can be determined which objects are completely invisible.

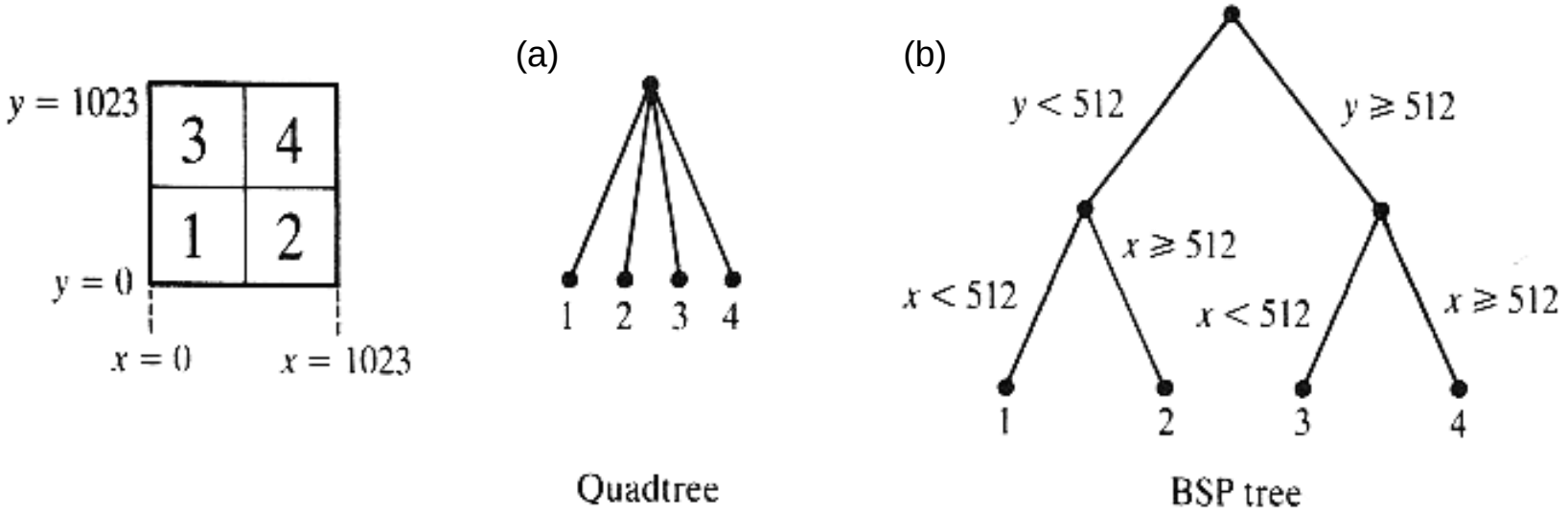
8.8 Space Partitioning

BSP trees (continued)

Example: sub-division of a 2-D scene

a) Using a quadtree

b) Using a BSP tree



8.8 Space Partitioning

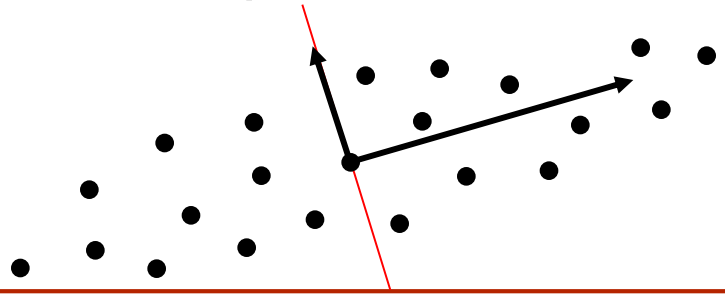
Principal Component Analysis (PCA)

An ideal choice of dividing planes for a BSP tree is offered by the **principal component analysis** (PCA). Let us assume that a complex scene is given by a point cloud

$$P_i \in IR^3 \quad (i=1,\ldots,n)$$

(for example object centers or vertices of polygons).

PCA defines an orthogonal coordinate system e_1, e_2, e_3 which orientation corresponds to the one of the point cloud.



8.8 Space Partitioning

Principal Component Analysis (continued)

Now, we choose the average of all points as the center of the coordinate system:

$$c = \frac{1}{n}\sum_{i=1}^{n} P_i$$

$$A = \begin{pmatrix} P_{1x} - c_x & P_{1y} - c_y & P_{1z} - c_z \\ P_{2x} - c_x & P_{2y} - c_y & P_{2z} - c_z \\ \vdots & \vdots & \vdots \\ P_{nx} - c_x & P_{ny} - c_y & P_{nz} - c_z \end{pmatrix}$$

$$B = \frac{1}{n-1} A^T A$$

$$b_{ij} = \frac{1}{n-1}\sum_{k=1}^{n} a_{ki} a_{kj}$$

B has the real eigenvalues λ_1, λ_2, λ_3 and eigenvectors e_1, e_2, e_3, i.e. $\lambda_i \cdot e_i = B \cdot e_i$. The eigenvectors in combination with the center c form the coordinate system we are looking for. The extent of the point cloud in direction of e_i is proportional to $\sqrt{\lambda_i}$.