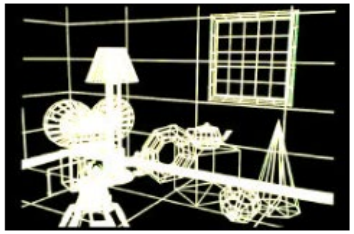
# CS5182 Computer Graphics
# Object Modeling

2024/25 Semester A

City University of Hong Kong (DG)



Scene description

Digital image

**Computer Graphics**

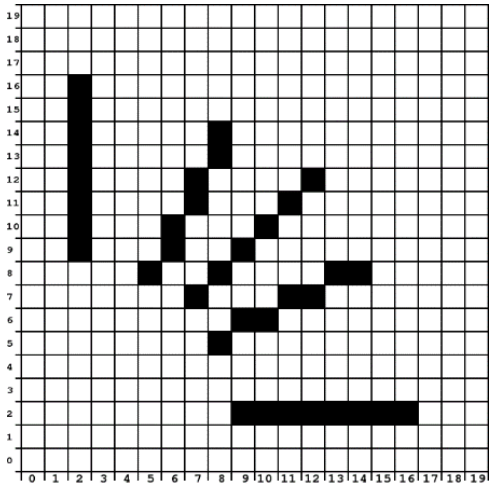
- Before we may produce images of a virtual environment, we need to construct ***geometry models*** to represent objects in the environment.

# Outline

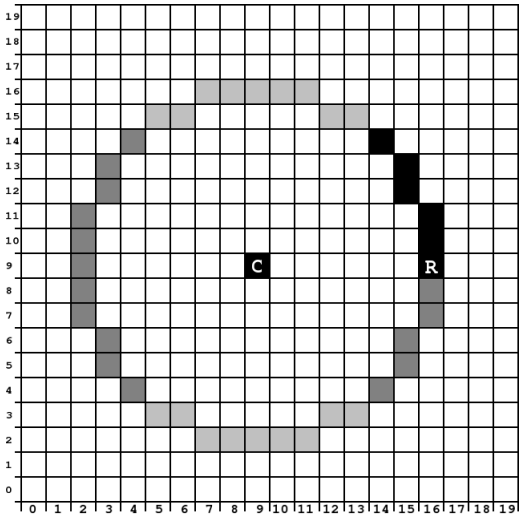
- 2D Drawing
- 3D Point Clouds
- Polygon Meshes
- Subdivision Surfaces
- Implicit Surfaces
- Parametric Surfaces
- Voxel Representation
- Constructive Solid Geometry (CSG)
- Fractals

# 2D Drawing

- Drawing an object involves converting the object into pixel pattern and updating the corresponding pixels in the frame buffer.



Thin line




Circle

# Drawing a Thin Line

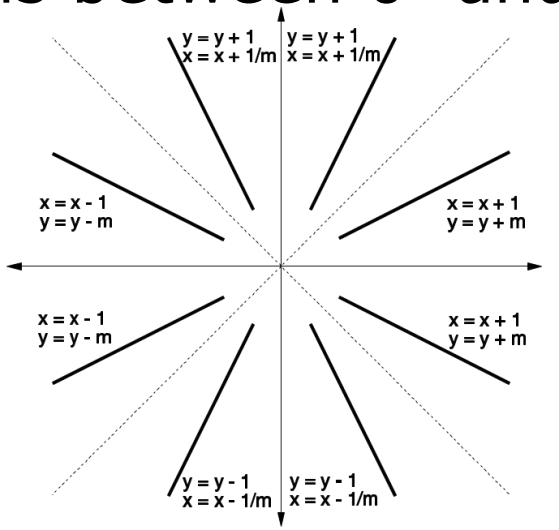
- Given two points $(x_1, y_1)$ and $(x_2, y_2)$, a line between them is defined as

$$y = mx + c$$

where $m = (y_2 - y_1)/(x_2 - x_1)$ is the slope of the line, and $c = y_1 - m \times x_1$ is a constant.

- We consider lines whose slopes are in the range of 0 and 1 (i.e., the slope of the line is between 0° and 45°.

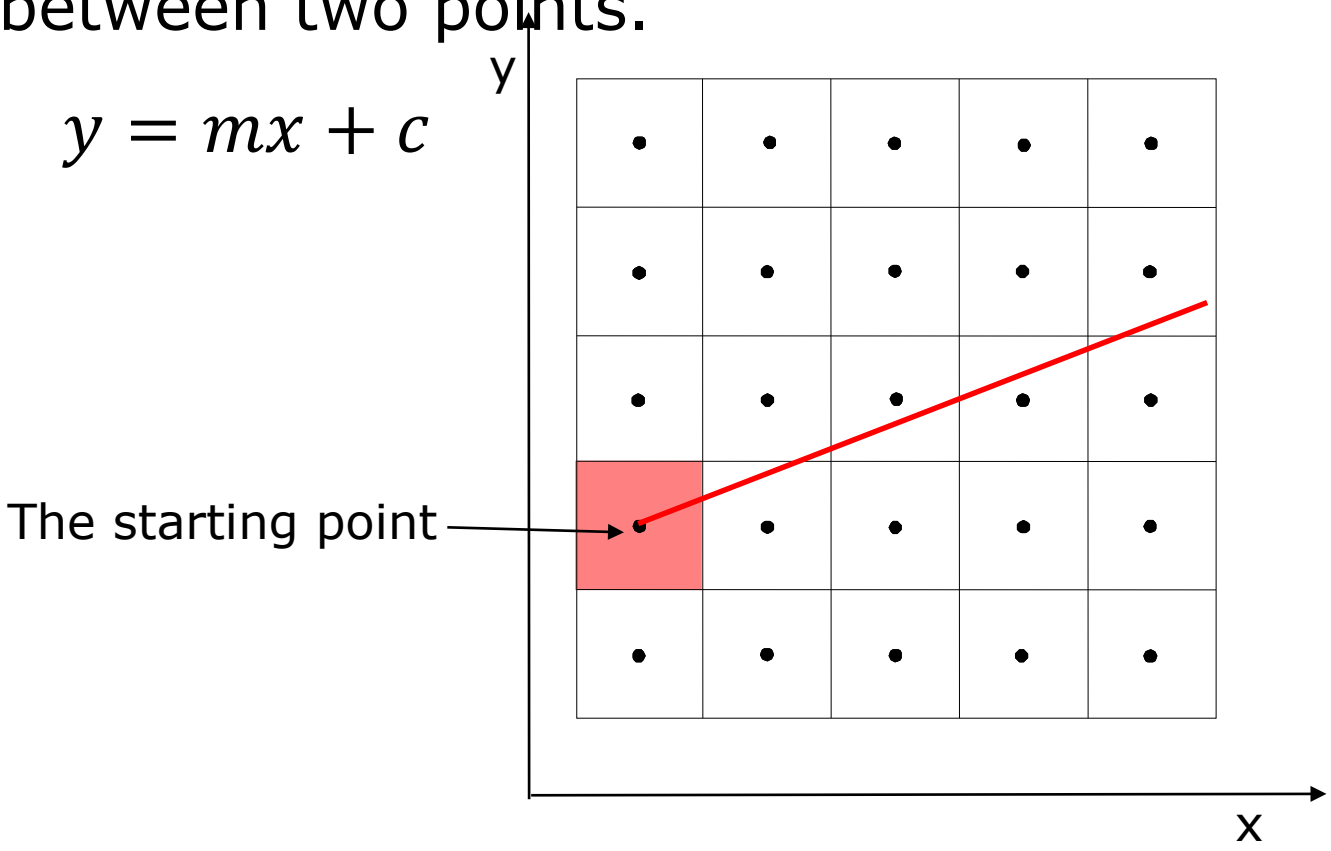
To consider lines of different slopes, we may swap $x$ and $y$ coordinates or change their signs in the above algorithm, thanks to the symmetrical property.



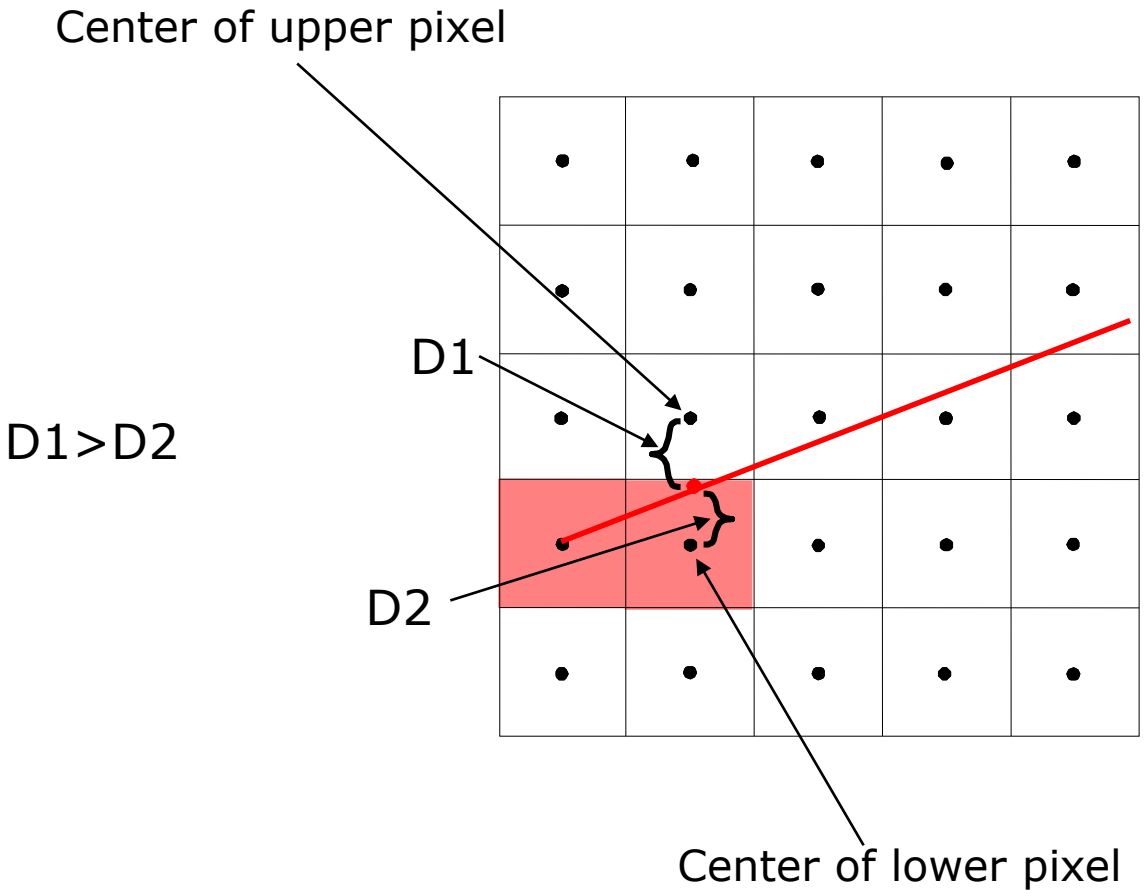
# Drawing a Thin Line

- The objective is to select pixels on the pixel grid to form a close approximation to a straight line between two points.

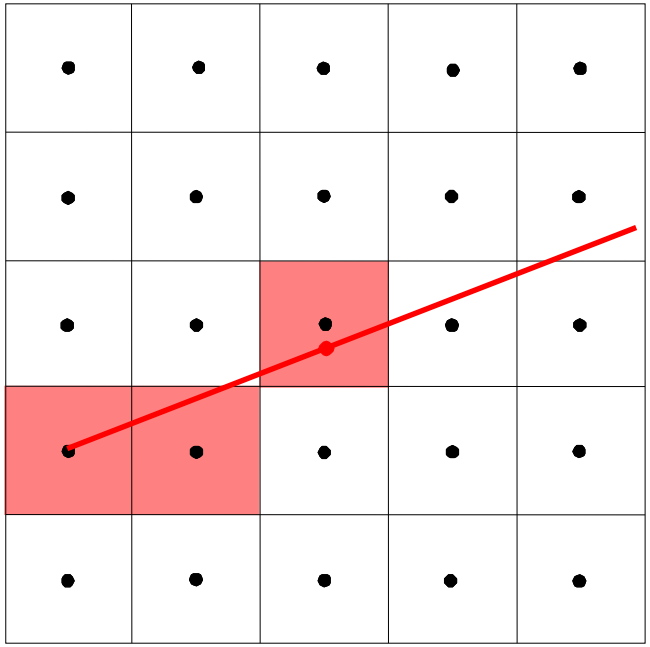
$$y = mx + c$$



# Drawing a Thin Line

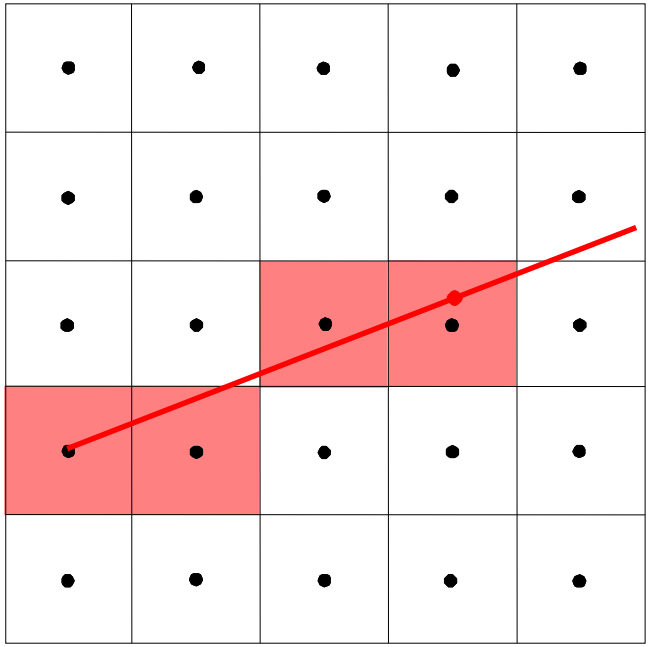


# Drawing a Thin Line



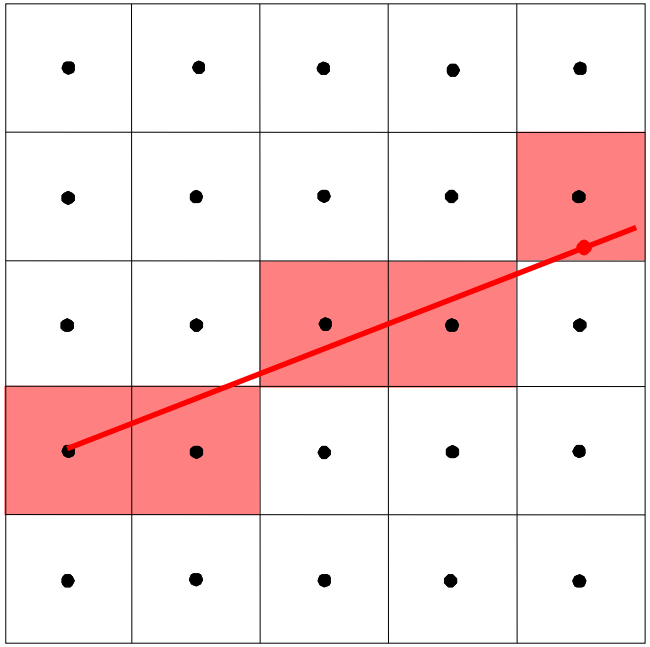
Center of lower pixel

# Drawing a Thin Line



Center of lower pixel

# Drawing a Thin Line



# Drawing a Thin Line

- 1) Set $x$ to $x_1$and $y$ to $y_1$, and shade this pixel;
- 2) Increase $x$ by 1, and correspondingly $y = y + m$
- 3) Compute D1 – distance of ($x$, $y$) from the center of the upper pixel;
- 4) Compute D2 – distance of ($x$, $y$) from the center of the lower pixel;
- 5) If D1 < D2 (i.e., the line is closer to the upper pixel), shade the upper pixel; otherwise, shade the lower pixel.
- 6) if the endpoint $(x_2, y_2)$ is not achieved, turn to 2); otherwise, stop.

```
void BresenhamLine(x1, y1, x2, y2)
int x1, y1, x2, y2;
{
    int dx, dy, x, y, xend, p. const1, const2;

    /* initialize variables */
    dx = Absolute(x1 - x2);          dy = Absolute(y1 - y2);
    const1 = 2 * dy;                 const2 = 2 * (dy - dx);       p = 2 * dy - dx;

    /* determine which point to use as the start, which as the end */
    if (x1 > x2) {
        x = x2;          y = y2;        xend = x1;
    } else {  /* ie. x1 <= x2 */
        x = x1;          y = y1;        xend = x2;
    }

    /* draw the line */
    SetPixel(x, y);
    while (x < xend) {
        x++;
        if (p < 0) {
            p += const1;
        } else {
            y++;          p += const2;
        }
        SetPixel(x, y);
    }
}
```

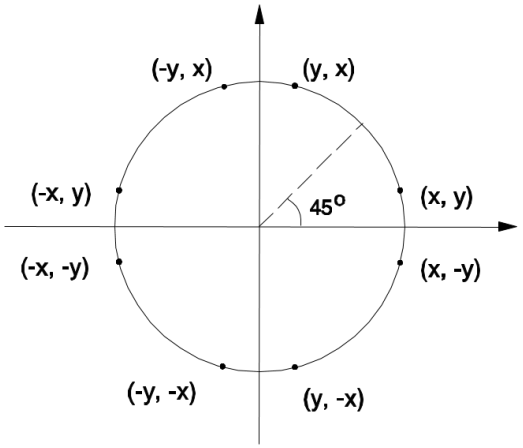
Developed in 1962 at IBM. The main idea of this algorithm is that it uses only integer arithmetic.

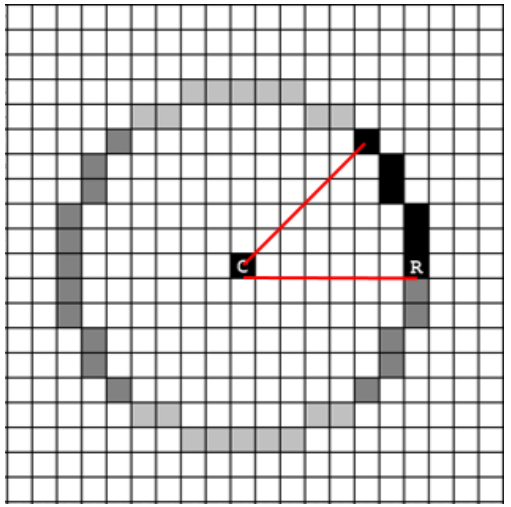
Please refer to pages 95–99 (Hearn and Baker) for the explanation of the algorithm.

# Drawing a Circle

- A circle can be specified as $(x - x_c)^2 + (y - y_c)^2 = r^2$ where $(x_c, y_c)$ is the center of the circle, and $r$ is the radius.

- The objective of the algorithm is to find a path through the pixel grid using pixels which are as close as possible to the solution of the circle equation.
  - At each step, the path is extended by choosing the adjacent pixel which minimizes $|(x - x_c)^2 + (y - y_c)^2 - r^2|$
  - To save time in drawing a circle, we can make use of the symmetrical property of a circle.



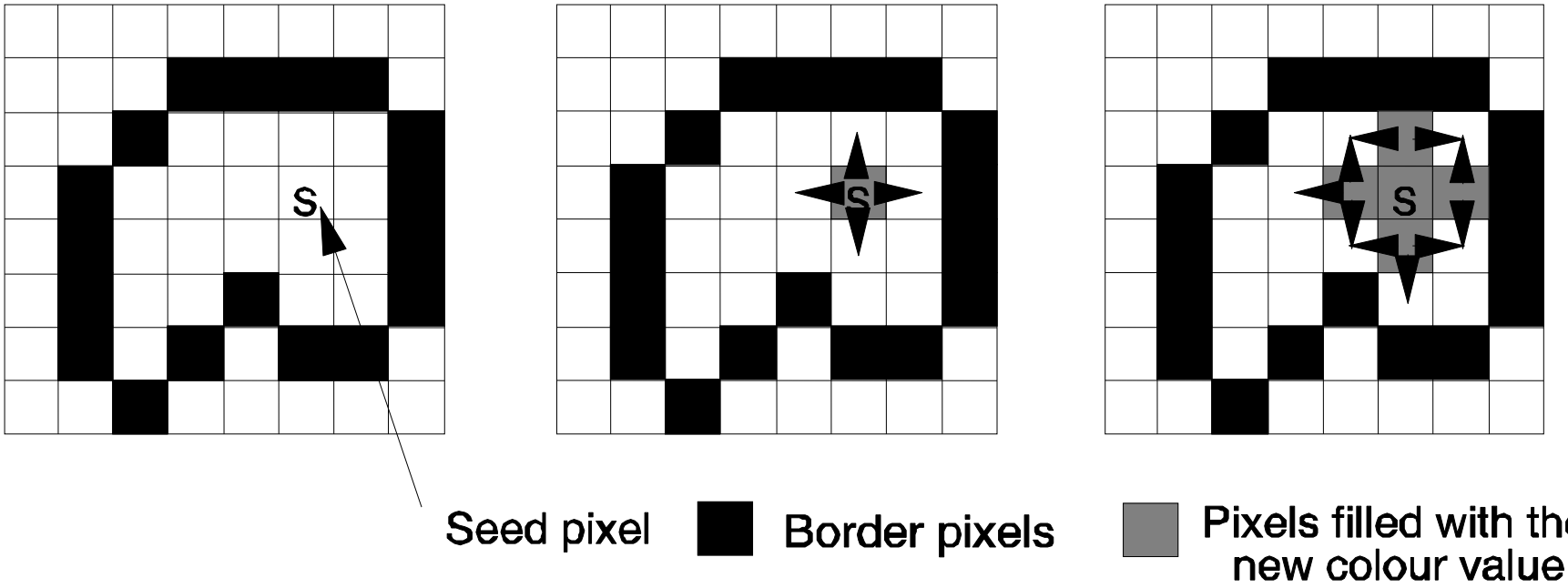
# Drawing a Circle

```
void BresenhamCircle(xc, yc, radius)
int xc, yc, radius;
{
   int x = 0;
   int y = radius;
   int p = 3 - 2 * radius;
   while (x < y) {
      PlotCirclePoints(xc, yc, x, y);
      if (p < 0) {
         p += (4 * x + 6);
      } else {
         p += (4 * (x - y) + 10);
         y--;
      }
      x++;
   }
   PlotCirclePoints(xc, yc, x, y);
}
```

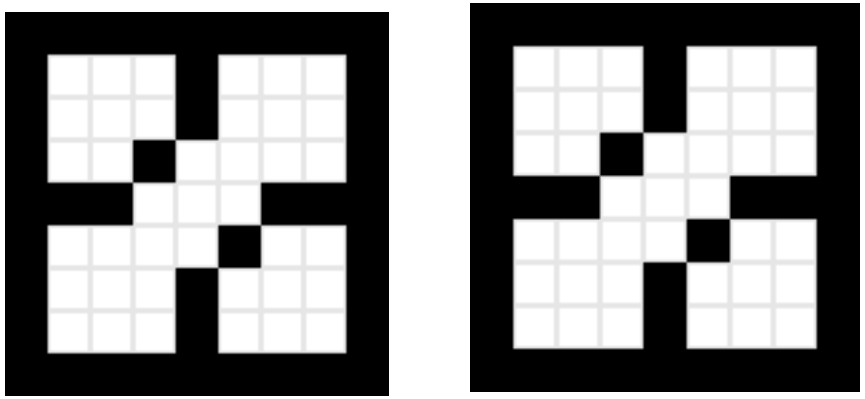
```
void PlotCirclePoints(xc, yc, x, y)
int xc, yc, x, y;
{
    SetPixel(xc + x, yc + y);
    SetPixel(xc - x, yc + y);
    SetPixel(xc + x, yc - y);
    SetPixel(xc - x, yc - y);
    SetPixel(xc + y, yc + x);
    SetPixel(xc - y, yc + x);
    SetPixel(xc + y, yc - x);
    SetPixel(xc - y, yc - x);
}
```

# Region Filling

- Flood-fill algorithm




Recursive flood fill with 4 directions



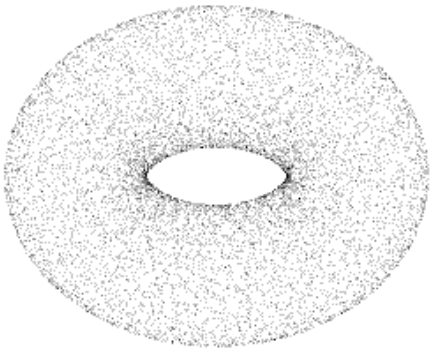
Recursive flood fill with 8 directions

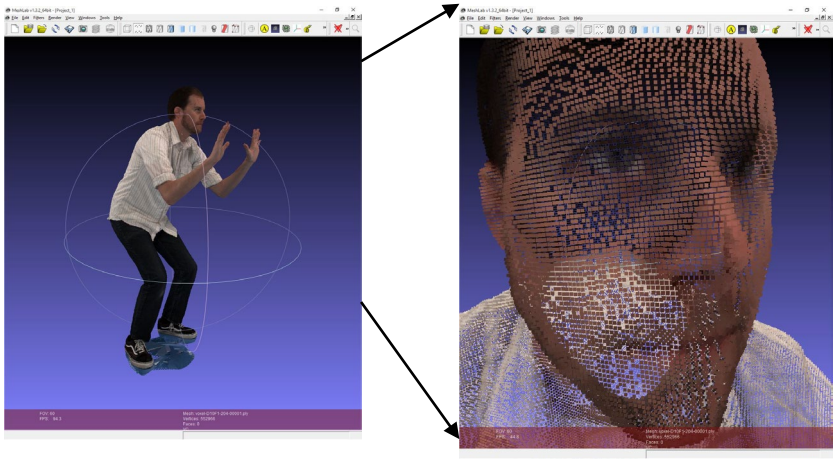
# Region Filling

- Flood-fill algorithm
  - The border pixels of the region are set to the border color.
  - A pixel which is completely inside the region is identified.
  - If the current pixel has a value the same as the border color or the fill color, we stop here.
  - Otherwise, we set the current pixel to the fill color, Usually this is the same as the border color.
  - We recursively set the left, right, top and bottom pixels as the current pixel and repeat the checking process.

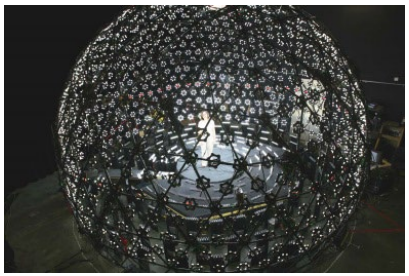
# 3D Point Clouds

- Unstructured set of 3D point samples
  - Each point consists of geometry information $(x, y, z)$ and other attributes, e.g., color $(r, g, b)$ and normal $(n_x, n_y, n_z)$



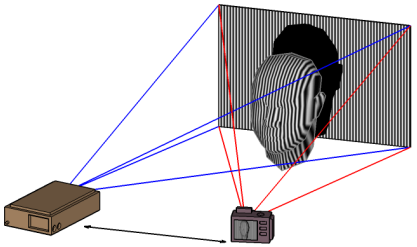


  - Acquired from range finder, computer vision, etc.



Multiview-based


Laser-based



Structured light-based

# 3D Point Clouds

- Widely used in autonomous driving
  - Lidar sensor which emits pulses of Infrared light and measures how long they take to come back after hitting nearby objects.

https://news.voyage.auto/an-introduction-to-lidar-the-key-self-driving-car-sensor-a7e405590cff?gi=1cac4c67174e

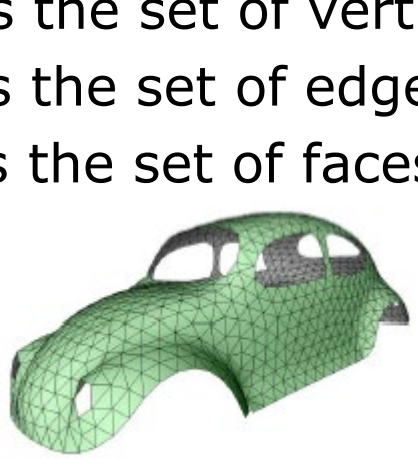
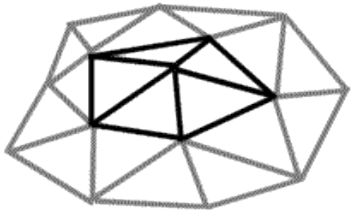
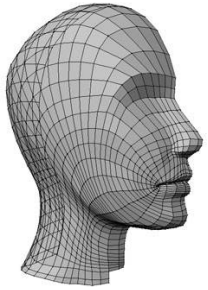
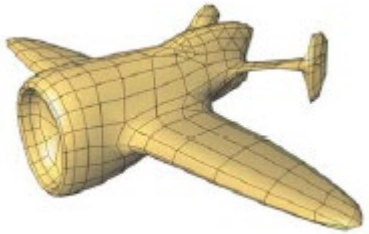
# 3D Point Clouds

- Advantage
  - Real-time acquisition
  - Connectivity/topological information-free
  - Suitable for real-time applications

- Disadvantage
  - Difficult to perform geometry computation

# Polygon Meshes

- A polygon mesh *M* is a 3-tuple (*V*, *E*, *F*) that defines the shape of a polyhedral object, where
  - *V* is the set of vertices
  - *E* is the set of edges
  - *F* is the set of faces

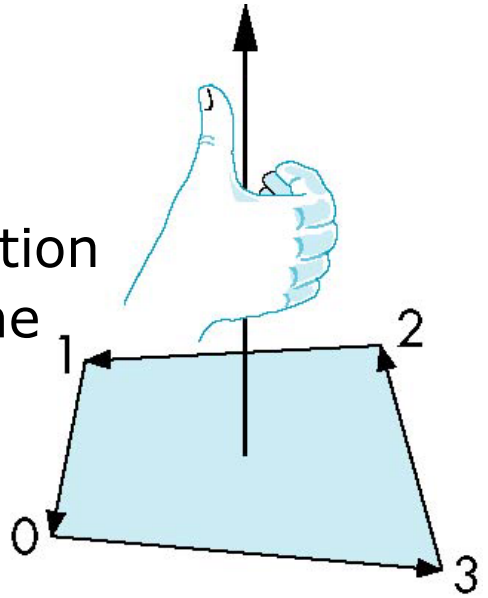





  - Each vertex must belong to at least one edge
  - Each edge must belong to at least one face
  - An edge is a **boundary** edge if it belongs to only one face

# Polygon Meshes

- Each face can be assigned an orientation by defining the ordering of its vertices
  - Orientation can be counter-clockwise or clockwise.

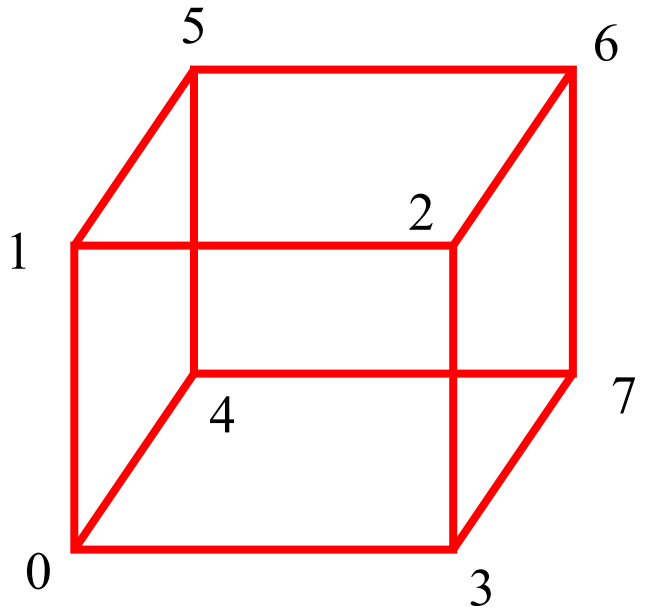
  - The orientation determines the normal direction of face. Usually, counterclockwise order is the "front" side.
  - The right-hand rule can be used to do this



# Polygon Meshes

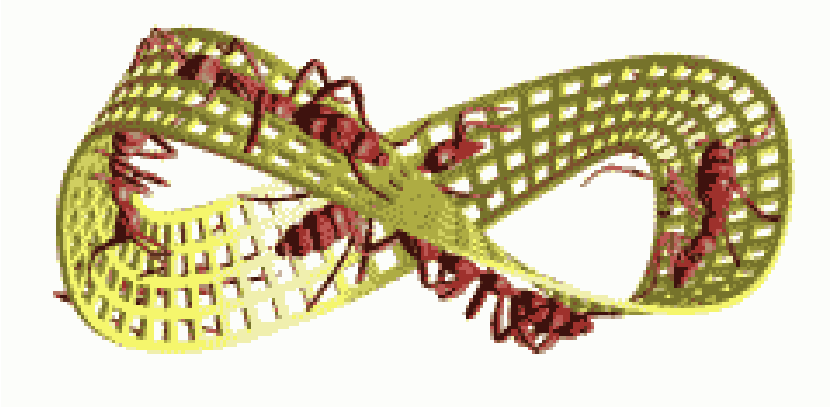
- Use the right-hand rule to indicate outward facing polygons, we may model a cube as follows:

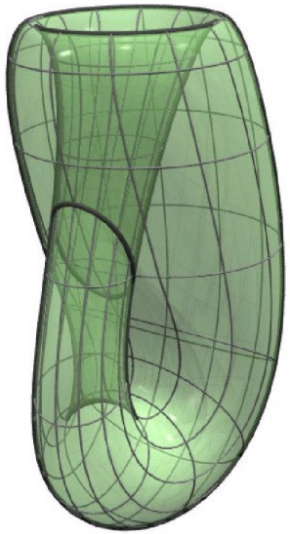
```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```



# Polygon Meshes

- A mesh is orientable if all faces can be oriented consistently (all CCW or all CW) such that each edge has two opposite orientations for its two adjacent faces.
- Not every mesh is orientable
  - Möbius strip
  - Klein bottle
  - …



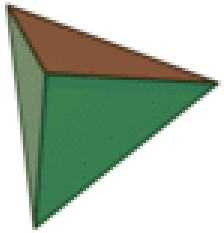


# Polygon Meshes

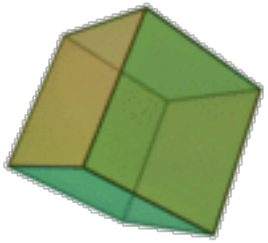
- Euler Formula
  - For a convex polyhedron (a **closed** manifold mesh **without** holes/handles), the relation between the numbers of vertices, edges, and faces is given by the Euler formula

$$|V| - |E| + |F| = 2$$

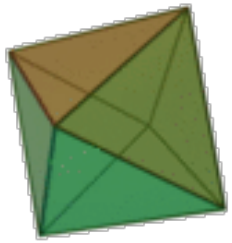
where $|V|, |E|$, and $|F|$ are the numbers of vertices, edges, and faces, respectively.



Tetrahedron
$|V| = 4$
$|E| = 6$
$|F| = 4$
4 - 6 + 4 = 2



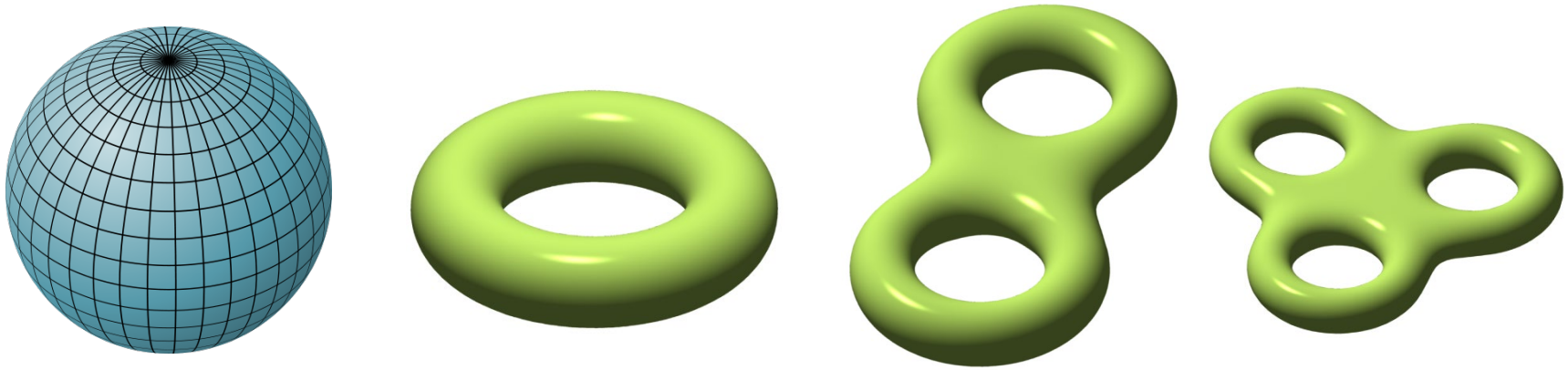
Cube
$|V| = 8$
$|E| = 12$
$|F| = 6$
8 - 12 + 6 = 2



Octahedron
$|V| = 6$
$|E| = 12$
$|F| = 8$
6 -12 + 8 = 2

# Polygon Meshes

- Genus
  - Defined as the **largest** number of **nonintersecting simple closed** curves that can be drawn on the surface **without separating it**.
  - Intuitively speaking, genus is the number of ”holes” in an orientable surface.



$g = 0$ $g = 1$ $g = 2$ $g = 3$

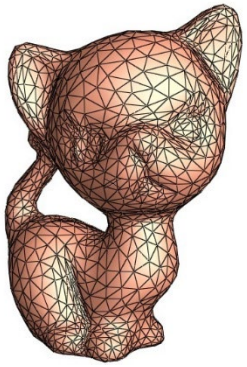
# Polygon Meshes

- For a **closed orientable** manifold mesh of genus ***g***, the relation between the number of vertices, edges, and faces is given by the Euler formula

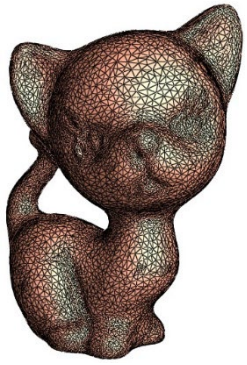
$$|V| - |E| + |F| = \chi = 2 - 2g$$

$\chi$ is called Euler characteristic.

$g$=1



|V|=1500
|E|=4500
|F|=3000



|V|=9000
|E|=27000
|F|=18000

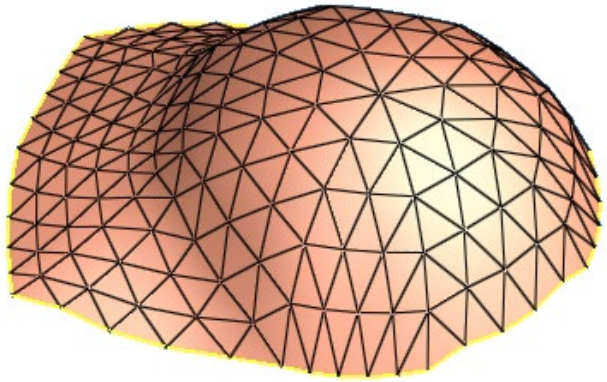
|V|=50000
|E|=150000
|F|=100000

$$|V| - |E| + |F| = 2 - 2g = 2 - 2 = 0$$

# Polygon Meshes

- For a **genus-*g* orientable** manifold mesh with ***b* boundaries**, the Euler formula is

$$|V| - |E| + |F| = 2 - 2g - b$$

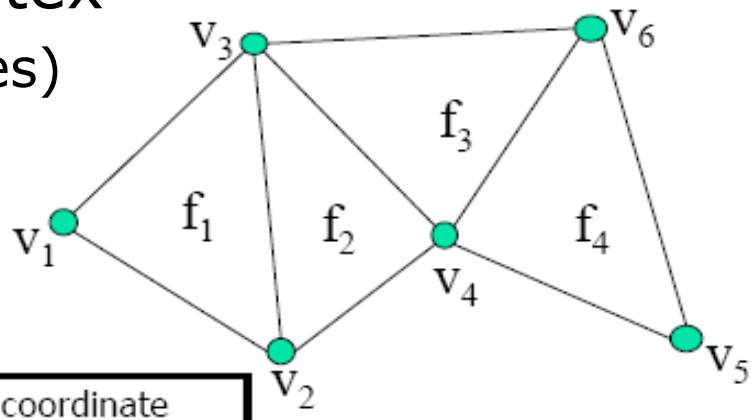


186 vertices;
325 faces;
510 edges;
1 boundary;
$g = 0$ (no handle)

$$186 - 510 + 325 = 1 = 2 - 2 \times 0 - 1$$

# Polygon Meshes

- Data Structure: face-vertex
  - List of vertices (coordinates)
  - List of faces




| vertex | coordinate |
|---|---|
| $v_1$ | $(x_1,y_1,z_1)$ |
| $v_2$ | $(x_2,y_2,z_2)$ |
| $v_3$ | $(x_3,y_3,z_3)$ |
| $v_4$ | $(x_4,y_4,z_4)$ |
| $v_5$ | $(x_5,y_5,z_5)$ |
| $v_6$ | $(x_6,y_6,z_6)$ |

| face | vertices (ccw) |
|---|---|
| $f_1$ | $(v_1, v_2, v_3)$ |
| $f_2$ | $(v_2, v_4, v_3)$ |
| $f_3$ | $(v_3, v_4, v_6)$ |
| $f_4$ | $(v_4, v_5, v_6)$ |

- There are other types of data structures: winged-edge, half-edge, quad-edge, corner-tables.

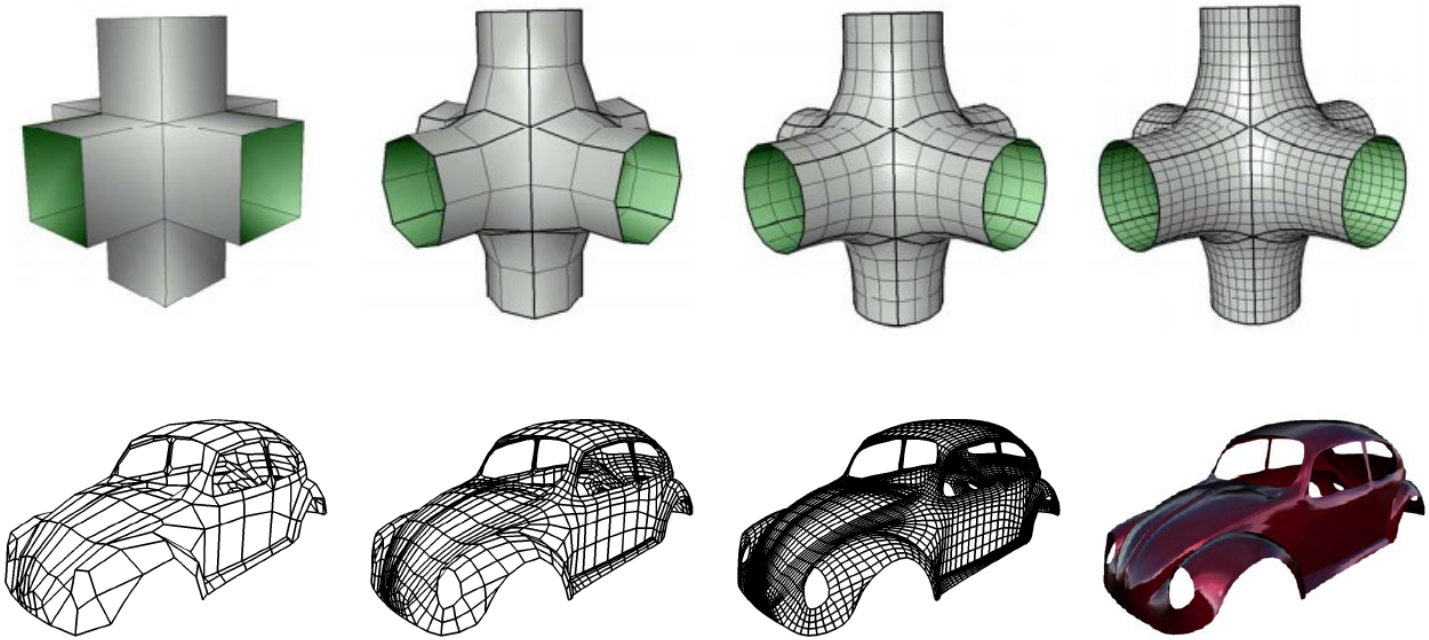
# Polygon Meshes

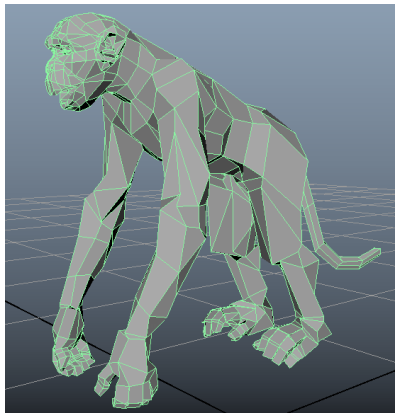
- Advantages
  - Arbitrary geometry
  - Arbitrary topology
  - Sharp features
  - Adaptive refinement
  - Efficient rendering

- Disadvantages
  - Smooth deformation is difficult.
  - Edges between polygons may become noticeable if near to the viewer.

# Subdivision Surfaces

- Subdivision defines a smooth surface as the limit of a sequence of successive refinements.





# Subdivision Surfaces

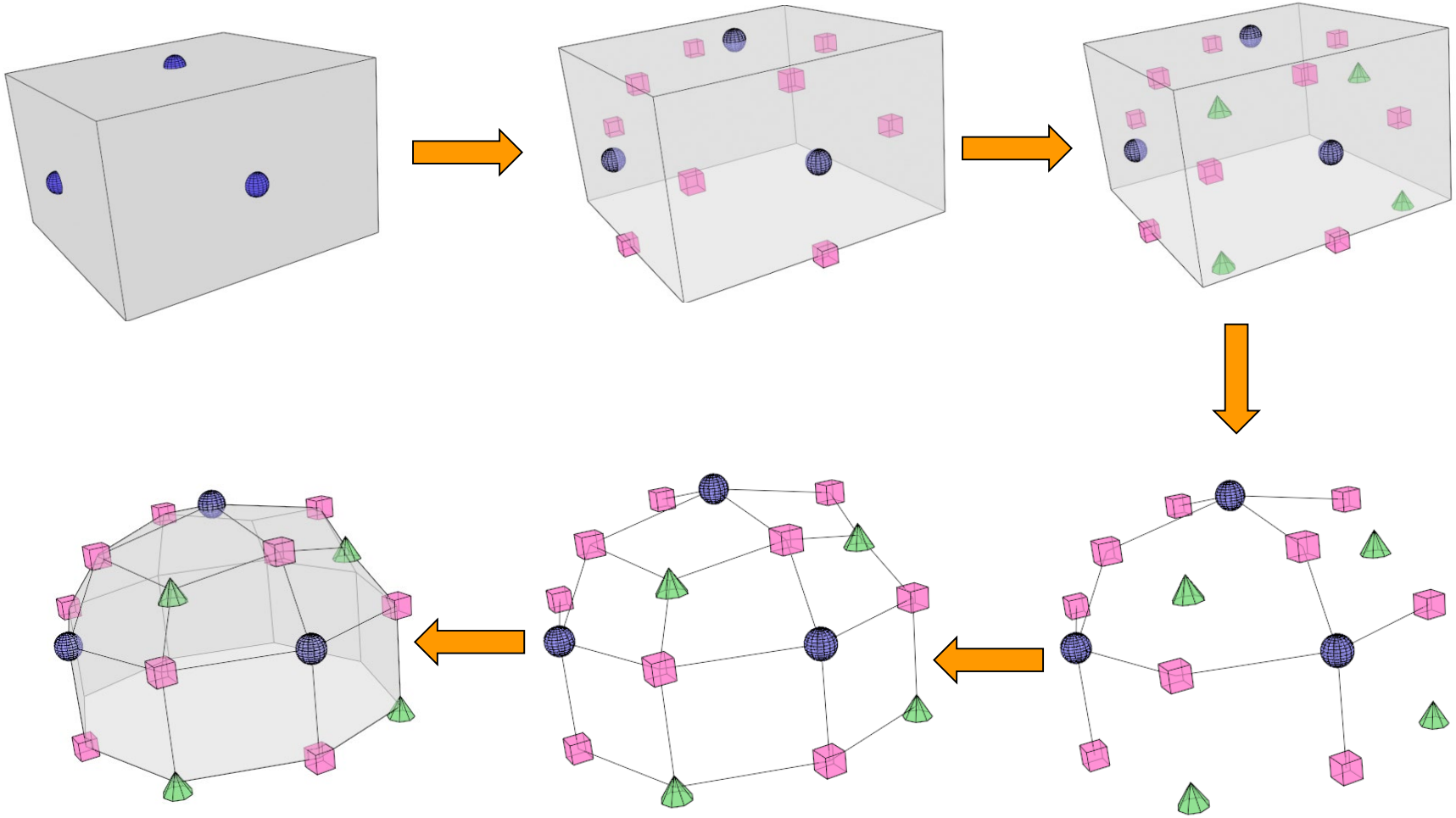
- Subdivision curve

Corner cutting

- The idea can be extended to surfaces, but it is more difficult.
  - How the connectivity changes
  - How the geometry changes
  - Catmull-Clark subdivision, Loop subdivision, Doo-Sabin subdivision, Butterfly subdivision

# Subdivision Surfaces

- Example: Catmull-Clark Subdivision Surface

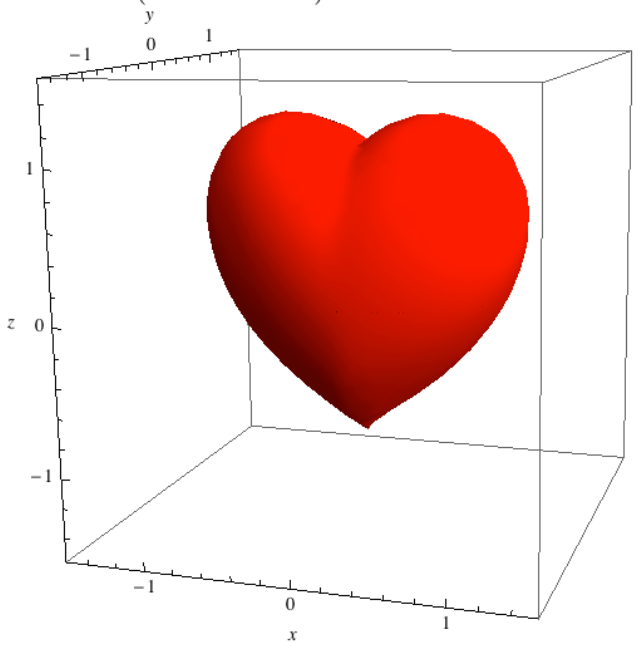


# Implicit Surfaces

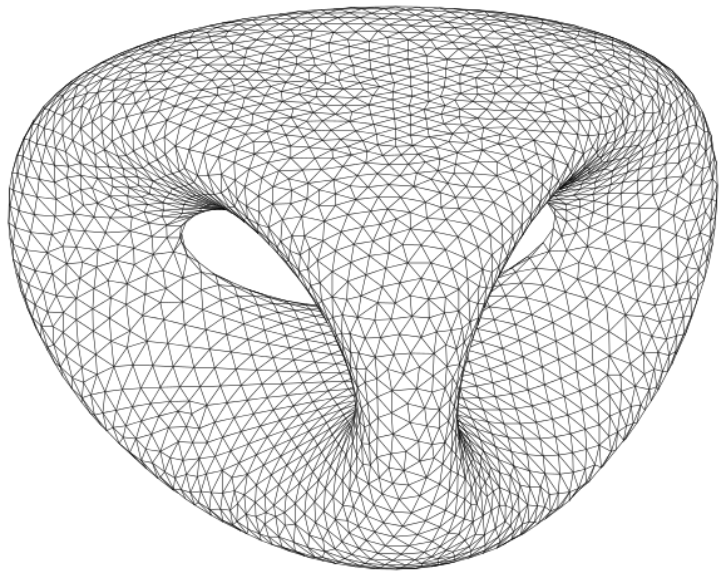
- Define the set of points on a surface by giving a procedure that can test to see if a point is on the surface. It is usually defined by an implicit function

$$f(x, y, z) = 0$$

$$\left(x^2 + \frac{9y^2}{4} + z^2 - 1\right)^3 - x^2 z^3 - \frac{9y^2 z^3}{200} = 0$$




$$2y(y^2 - 3x^2)(1 - z^2) + (x^2 + y^2)^2 - (9z^2 - 1)(1 - z^2) = 0$$



# Implicit Surfaces

- Advantages
  - Efficient check whether point is inside

$$f(x,y,z) < 0 \text{ (inside)}$$

$$f(x,y,z) = 0 \text{ (surface)}$$

$$f(x,y,z) > 0 \text{ (outside)}$$

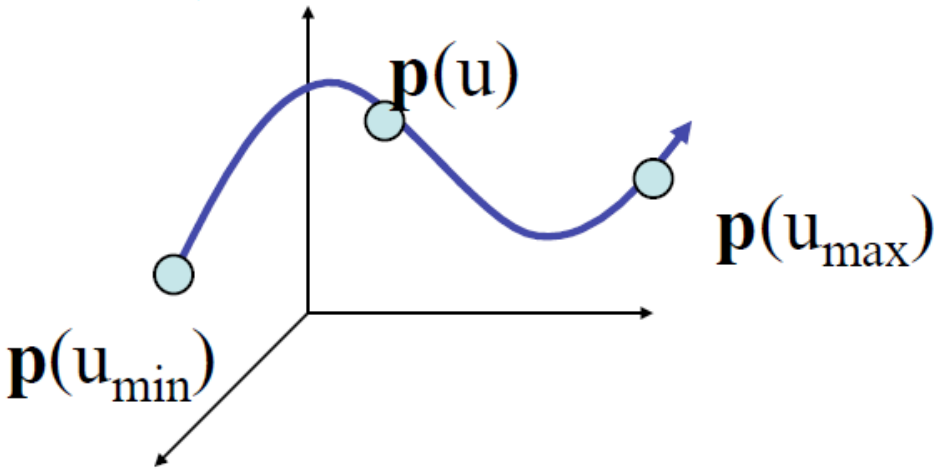
- Disadvantages
  - Does not allow us to easily generate a point on the surface.
  - Not easy to piecewise join them and maintain continuity and smoothness.
  - It is not easy to represent any bounded portion of an implicit surface.

# Parametric Surfaces

- Parametric curves: separate parametric equation for each spatial variable

$$\begin{cases} x = f_x(u) \\ y = f_y(u) \\ z = f_z(u) \end{cases} \qquad \tilde{\mathbf{P}}(u) = [x(u), y(u), z(u)]^T$$

  - For $u_{max} \geq u \geq u_{min}$ we trace out a curve in two or three dimensions



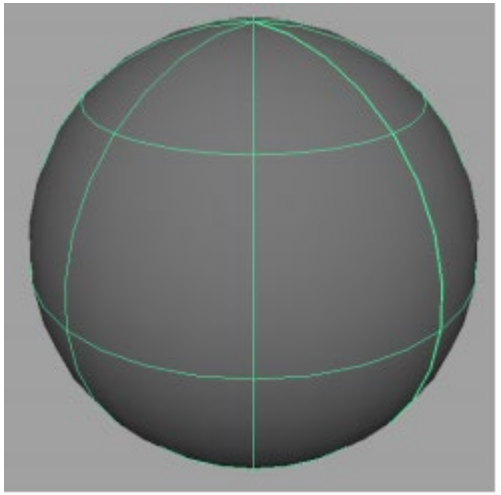
# Parametric Surfaces

- Surfaces can be represented parametrically

$$\begin{cases} x = f_x(s,t) \\ y = f_y(s,t) \\ z = f_z(s,t) \end{cases} \qquad \tilde{\mathbf{P}}(s,t) = [x(s,t), y(s,t), z(s,t)]^T$$

  - Example: sphere of radius $r$ centered at origin

$$\begin{cases} x = r\cos(s)\cos(t) \\ y = r\sin(s)\cos(t) \\ z = r\sin(t) \end{cases}$$



# Parametric Surfaces

- Parametric Curves (Cubic Parametric Polynomials)

$$\widetilde{\mathbf{P}}(u) = \mathbf{U} \times \mathbf{M} \times \mathbf{P}$$

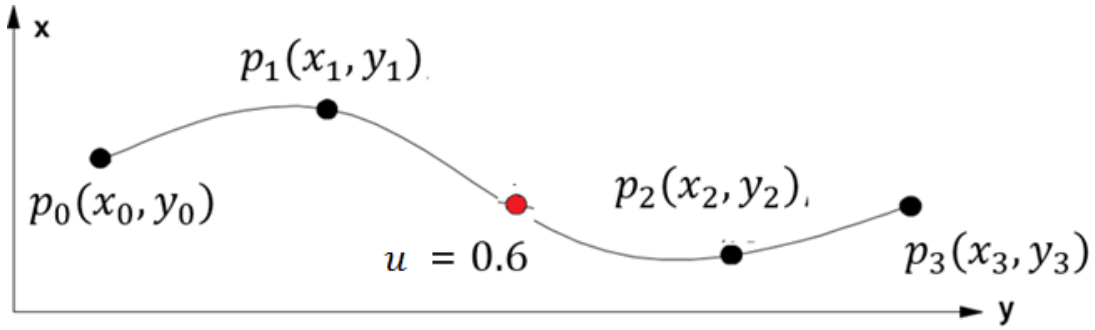
  - **U** is a $1 \times 4$ matrix of parameter *u* in the form of $\mathbf{U} = [u^3 \ u^2 \ u \ 1]$ where 0<= *u* <= 1
  - **P** is a $4 \times 1$ matrix called *geometry matrix:* $\mathbf{P} = [p_0 \ p_1 \ p_2 \ p_3]$. It is a set of control points for manipulating the shape of the curve.
  - **M** is a $4 \times 4$ matrix called *basis matrix*. It defines the basic properties of the curve, such as type of curve, affected region of each control point, and whether the curve passes through the control points at the two ends.

# Parametric Surfaces

- Parametric Curves (Cubic Parametric Polynomials)
  - Let $p_0(x_0, y_0)$, $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, and $p_3(x_3, y_3)$ be four control points. Given the matrix values of *M*, we may determine individual points (i.e., *x* and *y* coordinates) on the curve simply by changing the value of $u$ to generate a curve.

$$x(u) = [u^3 \;\; u^2 \;\; u^1 \;\; 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$y(u) = [u^3 \;\; u^2 \;\; u^1 \;\; 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$



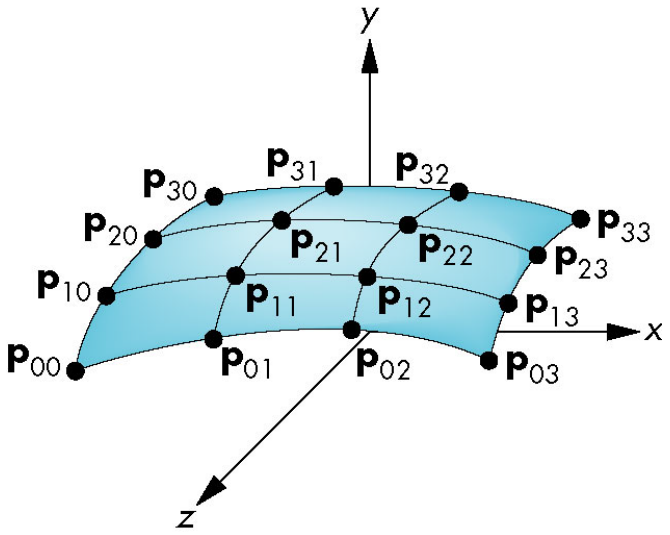
# Parametric Surfaces

- Parametric Surfaces

$$\tilde{\mathbf{P}}(s,t) = \mathbf{S} \times \mathbf{M} \times \mathbf{P} \times \mathbf{M}^T \times \mathbf{T}^T$$

  - **S** and **T** are $1 \times 4$ matrix of parameter *s* and *t* in the form of **S** = [$s^3$ $s^2$ $s$ 1] and **T** = [$t^3$ $t^2$ $t$ 1] with 0<= s, *t* <= 1.
  - **P** is a $4 \times 4$ matrix called *geometry matrix.* It is a set of control points for manipulating the shape of the curve.
  - **M** is a $4 \times 4$ matrix called *basis matrix*.

    It defines the basic properties of the curve, such as type of curve, affected region of each control point, and whether the curve passes through the control points at the two ends.

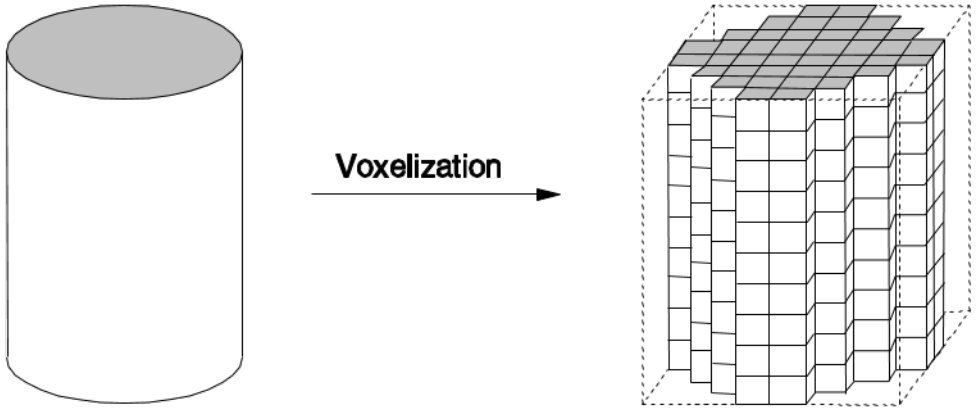


# Parametric Surfaces

- Advantages
  - Smooth deformation can easily be done by modifying the control points.
- Disadvantages
  - Parametric surfaces are expensive to render (evaluating the parametric equation once for each combination of *s* and *t* values).

# Voxel Representation

- In voxel representation, an object is decomposed into identical cells arranged in a fixed regular 3D grid.




  - A "1" may be used to indicate inside of the cylinder while a "0" may indicate outside of it.
  - Alternatively, we may use 8 bits to indicate object transparency. If a voxel stores a value of "0", it is fully-transparent. If it stores a "255", it is fully opaque. Any value between "0" and "255" indicates the opacity (or density) of the cell.

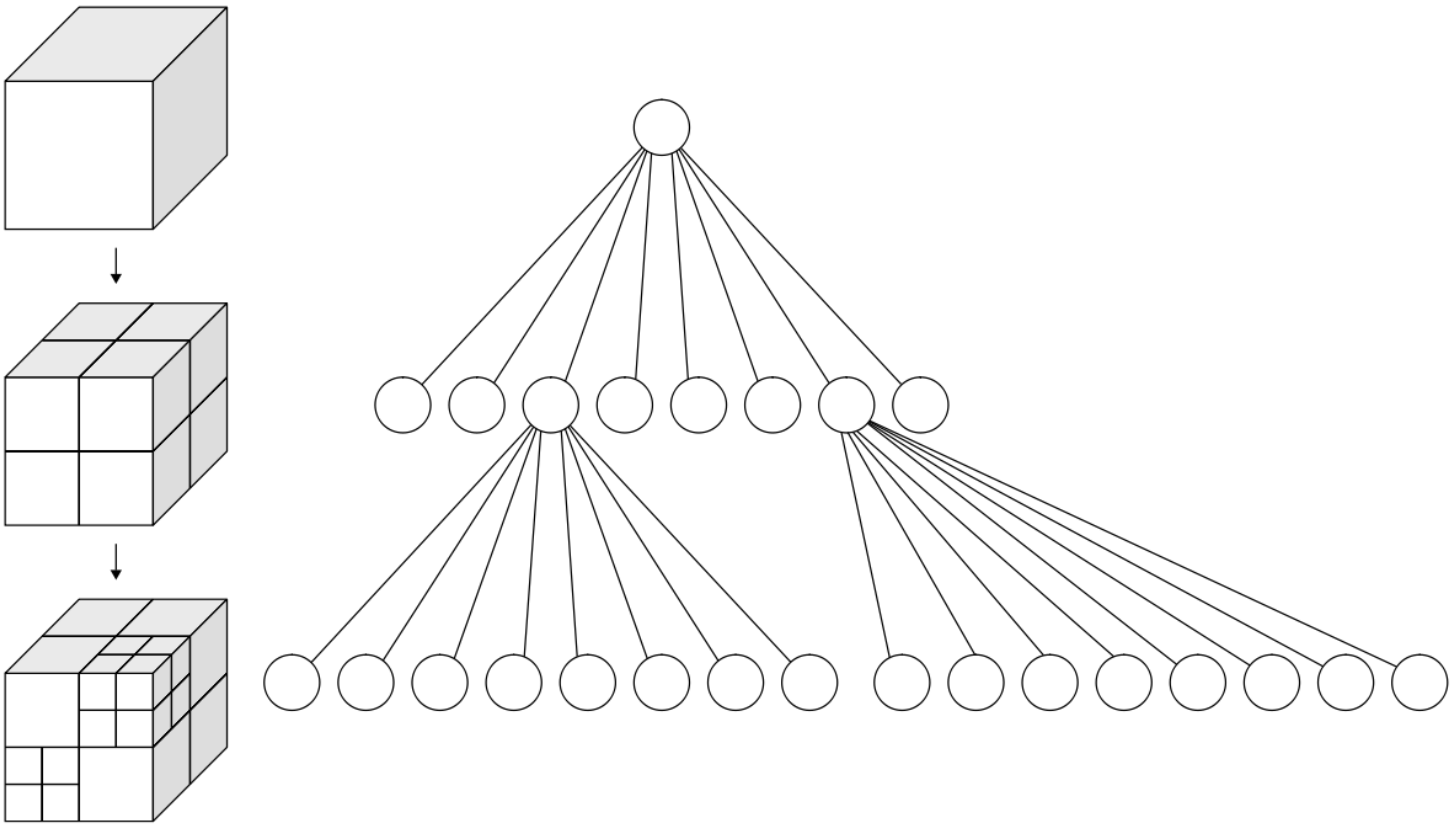
# Voxel Representation

- Advantages
  - Rendering the object is simple.

- Disadvantages
  - It is expensive to do scaling and deformation.
  - It usually requires a lot of memory to store the voxel data.

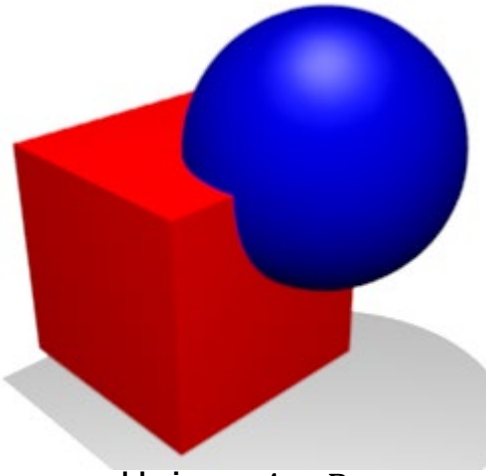
# Voxel Representation

- Octree representation may be used to save memory

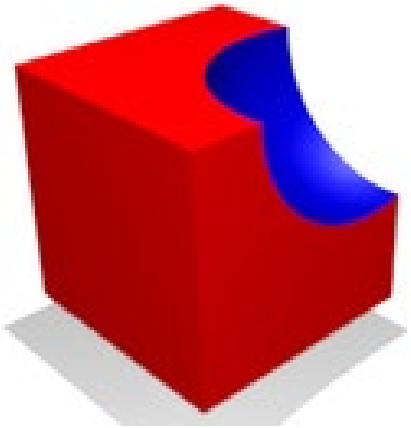


# Constructive Solid Geometry (CSG)

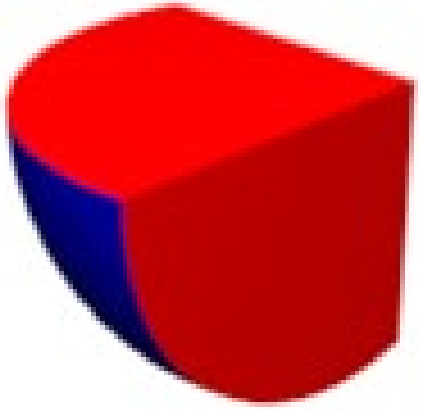
- Create a complex solid object by using Boolean operations to combine simpler objects.
  - Boolean operations for combining primitive objectives:



Union: $A \cup B$
Merger of two objects into one



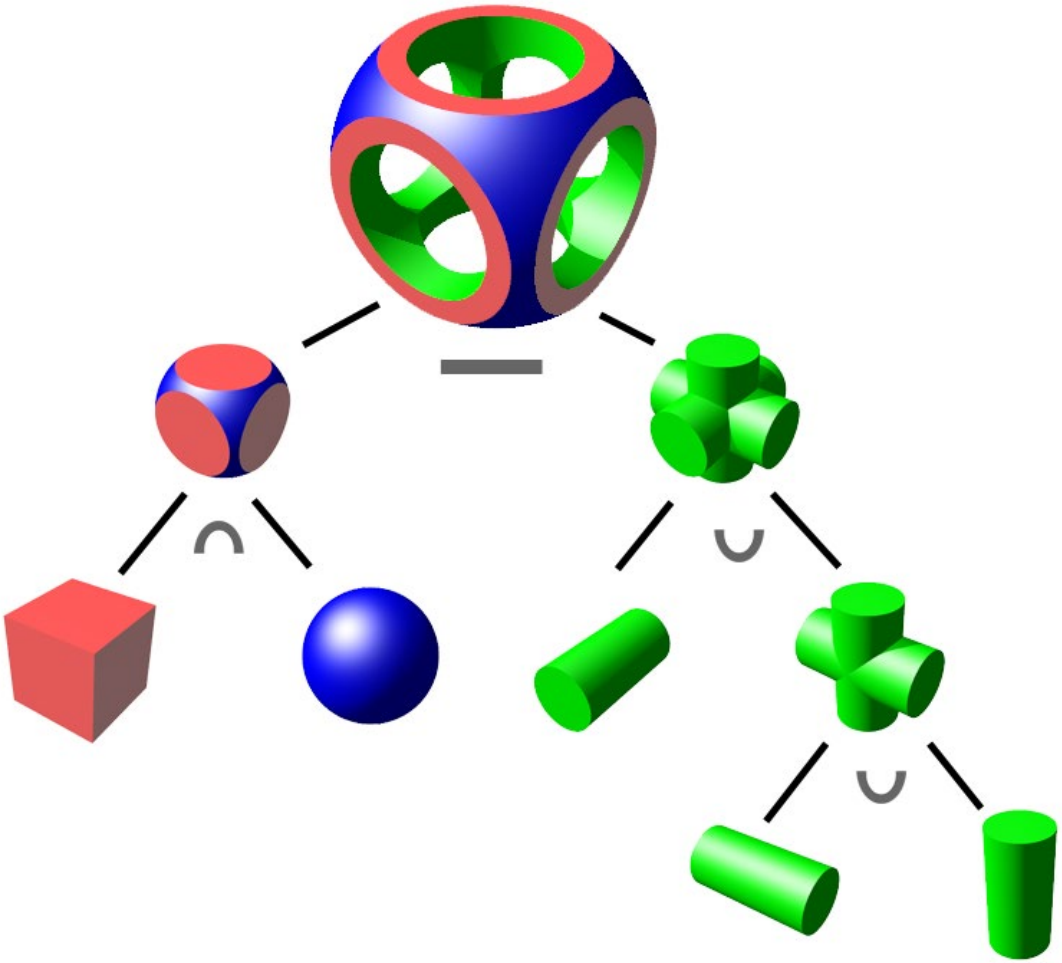
Difference: $A - B$
Subtraction of one object from another



Intersection: $A \cap B$
Portion common to both objects

# Constructive Solid Geometry (CSG)

- An example showing how a complex object can be created or represented using CSG.



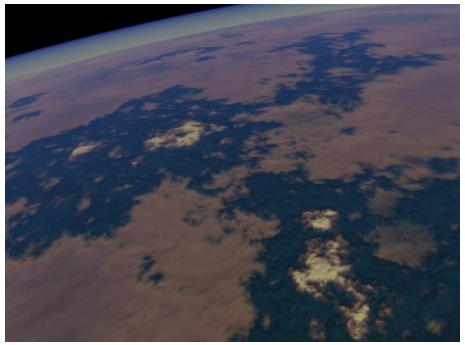
# Constructive Solid Geometry (CSG)

- Advantages
  - Complex objects can be created by combining simple ones.
  - It is easy to undo operations, given the CSG tree.

- Disadvantages
  - Whenever there is a modification to the object, all the operations in the CSG tree need to be evaluated again, unless some form of caching is performed.

# Fractals

- Fractal objects refer to those objects which are self-similar at all resolutions.
  - Most of the natural objects, such as trees, mountains and coastlines are considered as fractal objects, because no matter how far or how close one looks at them, they always appear to be somewhat similar.

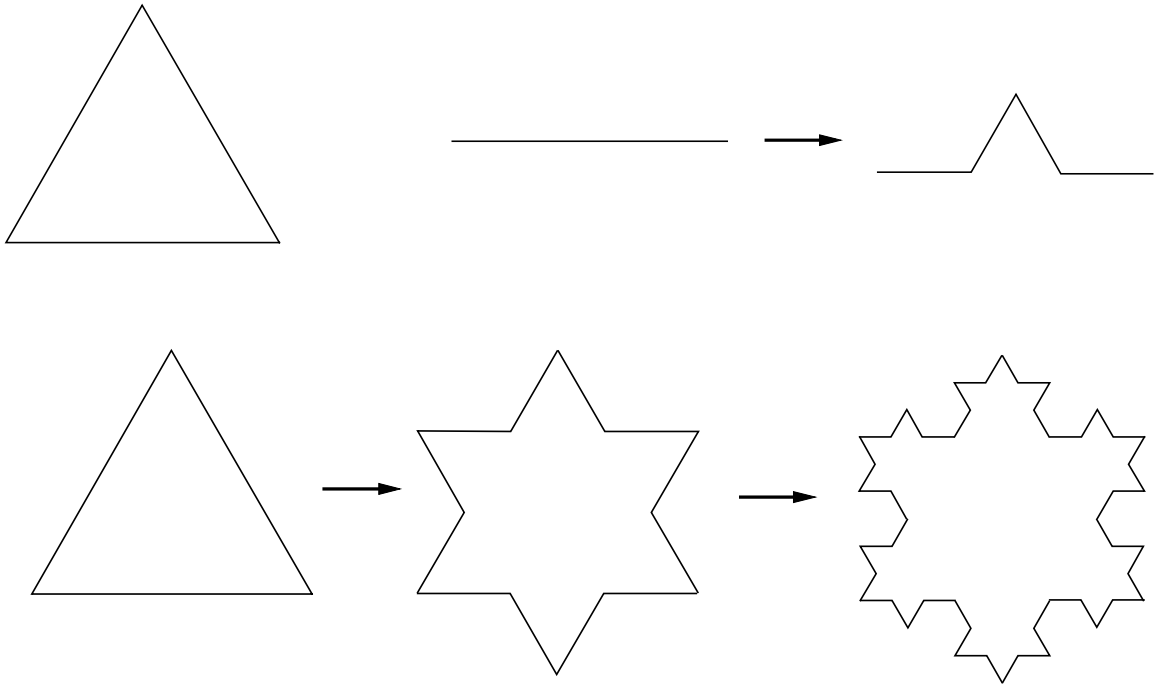


# Fractals

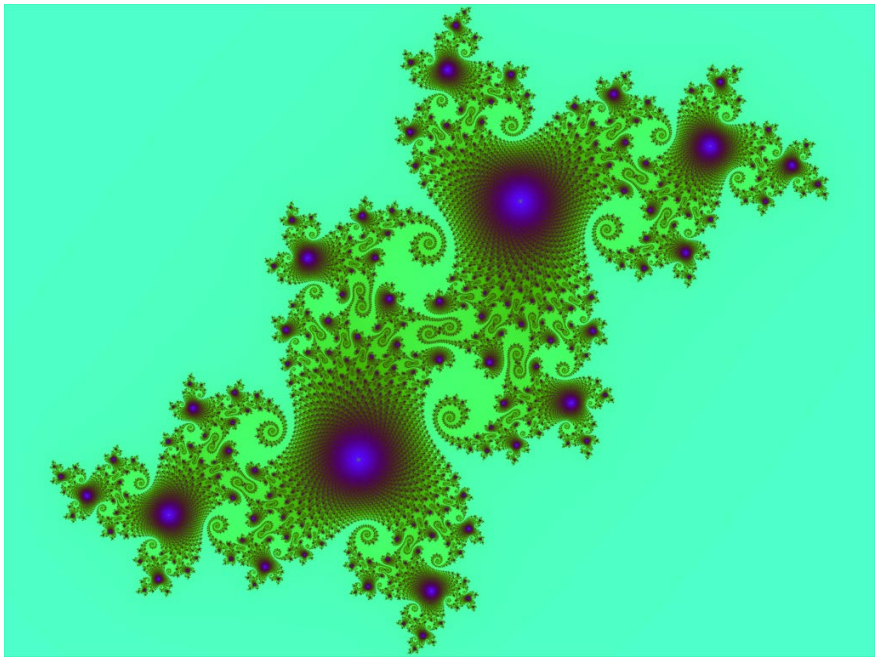
- A fractal object can be generated by recursively applying the same transformation function on a given object:

$$x = f(\ldots f(x))$$

  - For example, given:

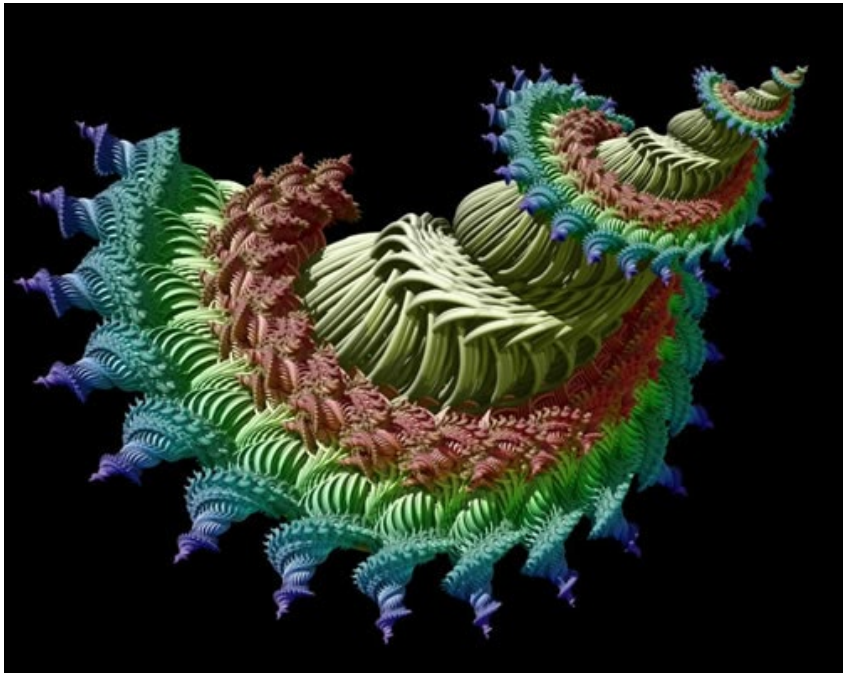


# Fractals

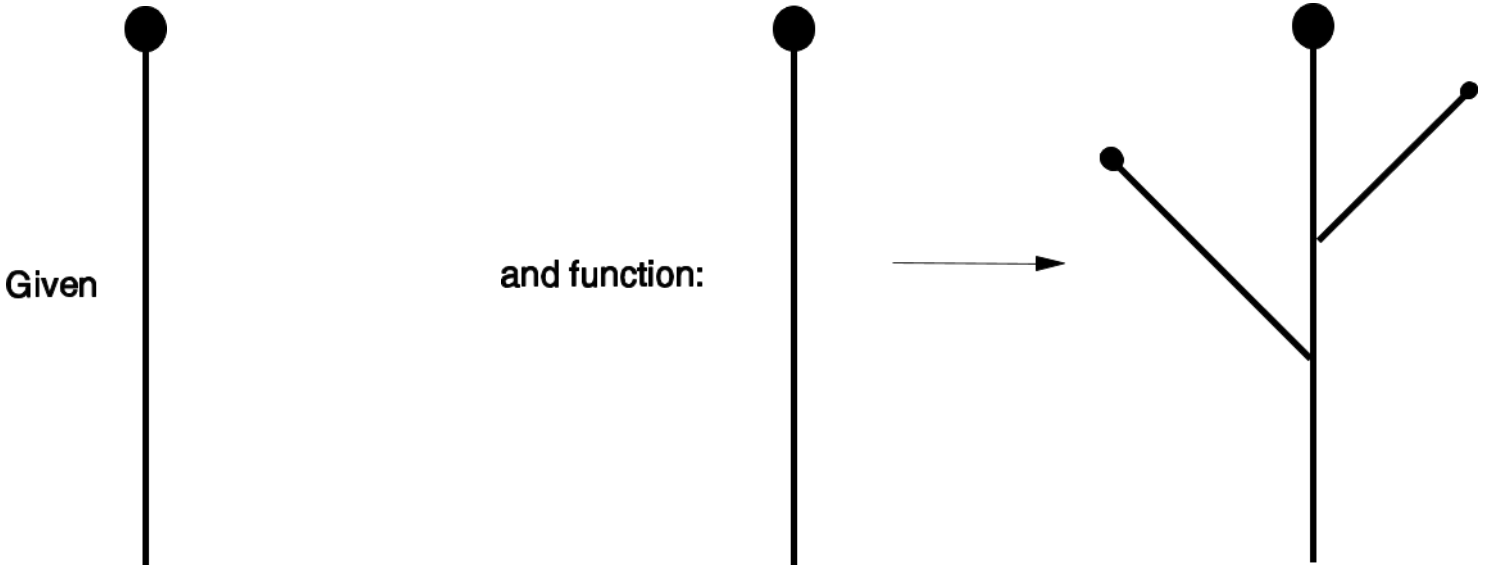


2D fractals

3D fractals



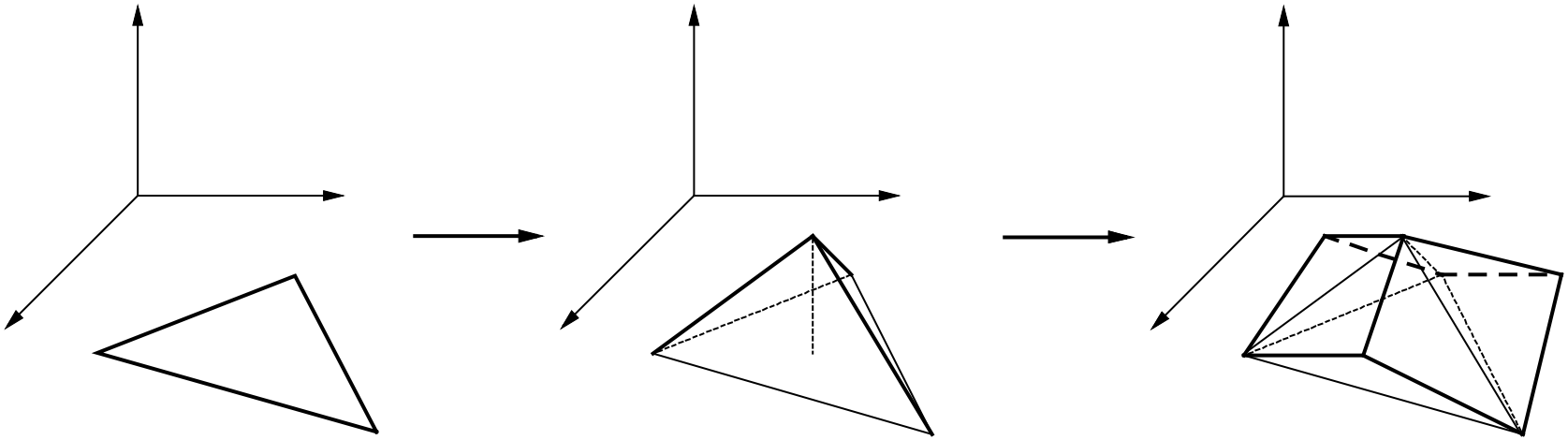
# Fractals




*What do we get?*

# Fractals

- We may also create 3D objects in a similar way by adding a third dimension and applying a recursive function to the input object.
  - Example: we replace each triangle of the input object with a pyramid, by inserting a single point at each step.



This will result in a mountain like object, with a regular appearance.

The main limitation of the above methods is that they generate exact detail of infinite resolution. This detail looks exactly the same at all resolutions. However, the details of real objects do not look exactly the same.

# Fractals

- For the generated objects to appear more realistic, we may introduce some random factors into the recursive functions, such as varying the width/height/position, etc.
  - Using the last example, if we insert each point at a somewhat random location, a more natural fractal mountain is resulted.