UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4370 Fall 2023

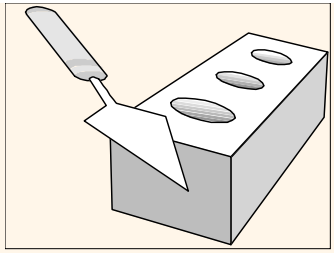
Interactive Computer Graphics

M & W 5:30 to 7:00 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE



COSC 4370

5:30 to 7

PLEASE

LOG IN

CANVAS

Please close all other windows.

NEXT.

11.08.2023 (W 5:30 to 7) (23)		**Lecture 11** **(Procedural Methods)**
11.13.2023 (M 5:30 to 7) (24)	Homework 8	**Lecture 12** **(Curves and Surfaces)**
11.15.2023 (W 5:30 to 7) (25)		**Lecture 13** **(Advanced Rendering)**
11.20.2023 (M 5:30 to 7) (26)		**PROJECT 4**
11.27.2023 (M 5:30 to 7) (27)		**EXAM 4 REVIEW**
11.29.2023 (M 5:30 to 7) (28)		**EXAM 4**
12.11.2023 (M 5:30 to 7)		**FINAL EXAM**

PROJECT 3

Publish | Edit | ⋮

The PROJECT is due by 5:30 PM of the due date class.

PROJECT 3.doc

(Please do not submit this file)

Submit:

1. *Your updated "Blueprint" matching the provided BUS Image* (-20 points)
2. *The zipped Visual Studio2019 or WEBGL "PROJECT 3".* (-30 points)
3. The "*PROJECT 3 Acceptance Testing*" (-20 points)
4. The "*PROJECT 3 Acceptance Testing Check Sheet*": (-10 points)
   - Rename it to score.OpenGL.docx. (MUST BE A .DOCX DOCUMENT)
   - Rename it to score.WEBGL.docx. (MUST BE A .DOCX DOCUMENT
5. *The ppt "PROJECT 3 PRESENTATION (no more than 20 slides)".* (-20 points)

Please have your presentation and your PROJECT 3 project code opened.

Unexcused absence or not responding when your name is called to present. (-20 points)

Please do not change the file names.

For the Grader: if the student did not submit, please skip and do not assign a ZERO (MISSING).

Points	100
Submitting	a file upload

Due	For	Available from	Until
Oct 30 at 5:30pm	Everyone	Oct 16 at 7pm	Oct 30 at 5:30pm

PROJECT 4

Publish | Edit

The PROJECT is due by 5:30 PM of the due date class.

PROJECT 4.doc

(Please do not submit this file)

Submit:

1. *The zipped Visual Studio2019 or WEBGL "PROJECT 4".* (-30 points)
2. *The "PROJECT 4 Acceptance Testing.docx".* (-20 points)
3. *The "PROJECT 4 Acceptance Testing Check Sheet.docx".* (-10 points)

   Rename it to score.OpenGL.docx. (MUST BE A .DOCX DOCUMENT)

   Rename it to score.WEBGL.docx. (MUST BE A .DOCX DOCUMENT)
4. *The ppt "PROJECT 4 PRESENTATION (no more than 20 slides)".* (-20 points)

Please have your presentation and your PROJECT 4 project code opened.

Unexcused absence or not responding when your name is called to present. (-20 points)

Please do not change the file names.

PROJECT 4
(100 points)

Hours spent:

The PROJECT is to be turned in before 5:30 PM of the due date class.

Submit:

1. *The zipped Visual Studio2019 or WEBGL "PROJECT 4".*
2. *The "PROJECT 4 Acceptance Testing".*
3. *The "PROJECT 4 Acceptance Testing Check Sheet" as "score.OPENGL.doc or score.WEBGL.doc" to CANVAS.*
4. *The ppt "PROJECT 4 PRESENTATION (no more than 20 slides)".*

S'COOL BUS - FALL 2023 Particle Systems Instructions
(The file "**S'COOL BUS FALL 2023**" contains the specifications for the BUS that you started modeling.)

1. The **glutCreateWindow** OpenGL call will take as parameter "**YourLastName** FirstName **Final BUS**".

2. Start with the **LastNameBUS.c** that you have created in PROJECT 3 and call the file **LastNameFinalBUS.c.**

3. Particle Systems:
Create 2 particle systems spheres, texture map them (happy faces??) and place them one in the **driver seat** and the other one in the **back of the bus centered**.

4. Navigating:
Navigate starting at DOOR towards your spheres with the intent to collide with them. When you collide with the spheres you can **simulate any desired effects like explosion**, etc. Successful collision with the 2 spheres will mark the completion of your project.

Get screenshots of the INSIDE of "YourLastName FirstName **Final BUS" for each**

PROJECT 4

Publish | Edit

The PROJECT is due by 5:30 PM of the due date class.

PROJECT 4.doc

(Please do not submit this file)

Submit:

1. *The zipped Visual Studio2019 or WEBGL "PROJECT 4".* (-30 points)
2. *The "PROJECT 4 Acceptance Testing.docx".* (-20 points)
3. *The "PROJECT 4 Acceptance Testing Check Sheet.docx".* (-10 points)

   Rename it to score.OpenGL.docx. (MUST BE A .DOCX DOCUMENT)

   Rename it to score.WEBGL.docx. (MUST BE A .DOCX DOCUMENT)
4. *The ppt "PROJECT 4 PRESENTATION (no more than 20 slides)".* (-20 points)

Please have your presentation and your PROJECT 4 project code opened.

Unexcused absence or not responding when your name is called to present. (-20 points)

Please do not change the file names.

Name: _____

PROJECT 4 Acceptance Testing:

Insert screenshots of the output produced and insert it in this file ……..

When done attach this file to "PROJECT 4" CANVAS assignment.

Particle Systems:

1. **Create particle system** in the **driver seat**

SCREENSHOT Output of your program is:

……..

2. **Create particle system** in the **back of the bus centered**

SCREENSHOT Output of your program is:

……..

Navigation:

1. **Start at the DOOR** looking toward the middle back BUS

SCREENSHOT Output of your program is:

……..

2. **Second, Navigate starting at DOOR towards your spheres** Driver Seat **with the intent to collide with it.**

Take some SCREENSHOTs Output of your program:

……..

3. **Third, Navigate starting at DOOR towards your spheres** Back of the Bus **with the intent to collide with it.**

Take some SCREENSHOTs Output of your program:

……..

Copy and Paste your .c (.cpp) file here:

……

PROJECT 4

Publish | Edit

The PROJECT is due by 5:30 PM of the due date class.

PROJECT 4.doc

(Please do not submit this file)

Submit:

1. *The zipped Visual Studio2019 or WEBGL "PROJECT 4".* (-30 points)
2. The "*PROJECT 4 Acceptance Testing.docx*". (-20 points)
3. Th "*PROJECT 4 Acceptance Testing Check Sheet.docx*". (-10 points)

   Rename it to score.OpenGL.docx. (MUST BE A .DOCX DOCUMENT)

   Rename it to score.WEBGL.docx. (MUST BE A .DOCX DOCUMENT)
4. *The ppt "PROJECT 4 PRESENTATION (no more than 20 slides)".* (-20 points)

Please have your presentation and your PROJECT 4 project code opened.

Unexcused absence or not responding when your name is called to present. (-20 points)

Please do not change the file names.

PROJECT 4 Hours:

Acceptance Testing Check Sheet:

CHECKLIST (YOU MUST SCORE YOURSELF!):

1. Submitted PROJECT4.zip? -30 points
2. Submitted Acceptance Testing.docx? -20 points
3. Submitted Acceptance Testing Check Sheet as score.OPENGL.docx or score.WEBGL.docx? -10 points
4. Submitted PROJECT 4 PRESENTATION.pptx? -20 points

Particle Systems:

1. ACCEPTANCE TESTING Step 1 20 points
2. ACCEPTANCE TESTING Step 2 20 points

Navigation:

3. ACCEPTANCE TESTING Step 1 20 points
4. ACCEPTANCE TESTING Step 2 20 points
5. ACCEPTANCE TESTING Step 3 20 points

Score

I CERTIFY THAT THIS IS MY OWN WORK and THE CHECKBOXES reflect the completion of these DELIVERABLES.

PROJECT 4

Publish | Edit

The PROJECT is due by 5:30 PM of the due date class.

PROJECT 4.doc

(Please do not submit this file)

Submit:

1. *The zipped Visual Studio2019 or WEBGL "PROJECT 4".* (-30 points)
2. The "*PROJECT 4 Acceptance Testing.docx*". (-20 points)
3. The "*PROJECT 4 Acceptance Testing Check Sheet.docx*". (-10 points)

   Rename it to score.OpenGL.docx. (MUST BE A .DOCX DOCUMENT)

   Rename it to score.WEBGL.docx. (MUST BE A .DOCX DOCUMENT)
4. The ppt "*PROJECT 4 PRESENTATION (no more than 20 slides)*". (-20 points)

Please have your presentation and your PROJECT 4 project code opened.

Unexcused absence or not responding when your name is called to present. (-20 points)

Please do not change the file names.

COSC 4370 – Computer Graphics

Lecture 11

Procedural Methods

Chapter 11

Procedural Methods

Thus far, we have assumed that the geometric objects that we wish to create can be described by their surfaces, and that these surfaces can be modeled (or approximated) by convex planar polygons. Our use of **polygonal objects** was dictated by the ease with which we could describe these **objects** and our ability to render them on existing systems. The success of computer graphics attests to the importance of such **models**.

Nevertheless, **even as these models were being used in large CAD applications, for flight simulators, in computer animations, in interactive video games, and to create special effects in films, both users and developers recognized their limitations**. **Physical objects such as clouds, smoke, and water did not fit this style of modeling. Adding physical constraints and modeling complex behaviors of objects were not part of polygonal modeling.**

In response to such problems, **researchers have developed procedural models, which use algorithmic methods to build representations of the underlying phenomena, generating polygons only as needed during the rendering process.**

3D Modeling

- **Polygonal meshes** **capture the shape** of **complex 3D objects** in simple data structures.
  - Platonic solids, the Buckyball, geodesic domes, prisms.
  - Extruded or swept shapes, and surfaces of revolution.
  - Solids with smoothly curved surfaces.

- **Animated Particle systems**: each **particle** responds to conditions.

- **Physically based systems**: the various **objects** in a scene are **modeled** as connected by springs, gears, electrostatic forces, gravity, or other mechanisms.

3D Modeling

- **Polygonal meshes** **capture the shape** of **complex 3D objects** in simple data structures.
  - Platonic solids, the Buckyball, geodesic domes, prisms.
  - Extruded or swept shapes, and surfaces of revolution.
  - Solids with smoothly curved surfaces.

- **Animated Particle systems**: each particle responds to conditions.

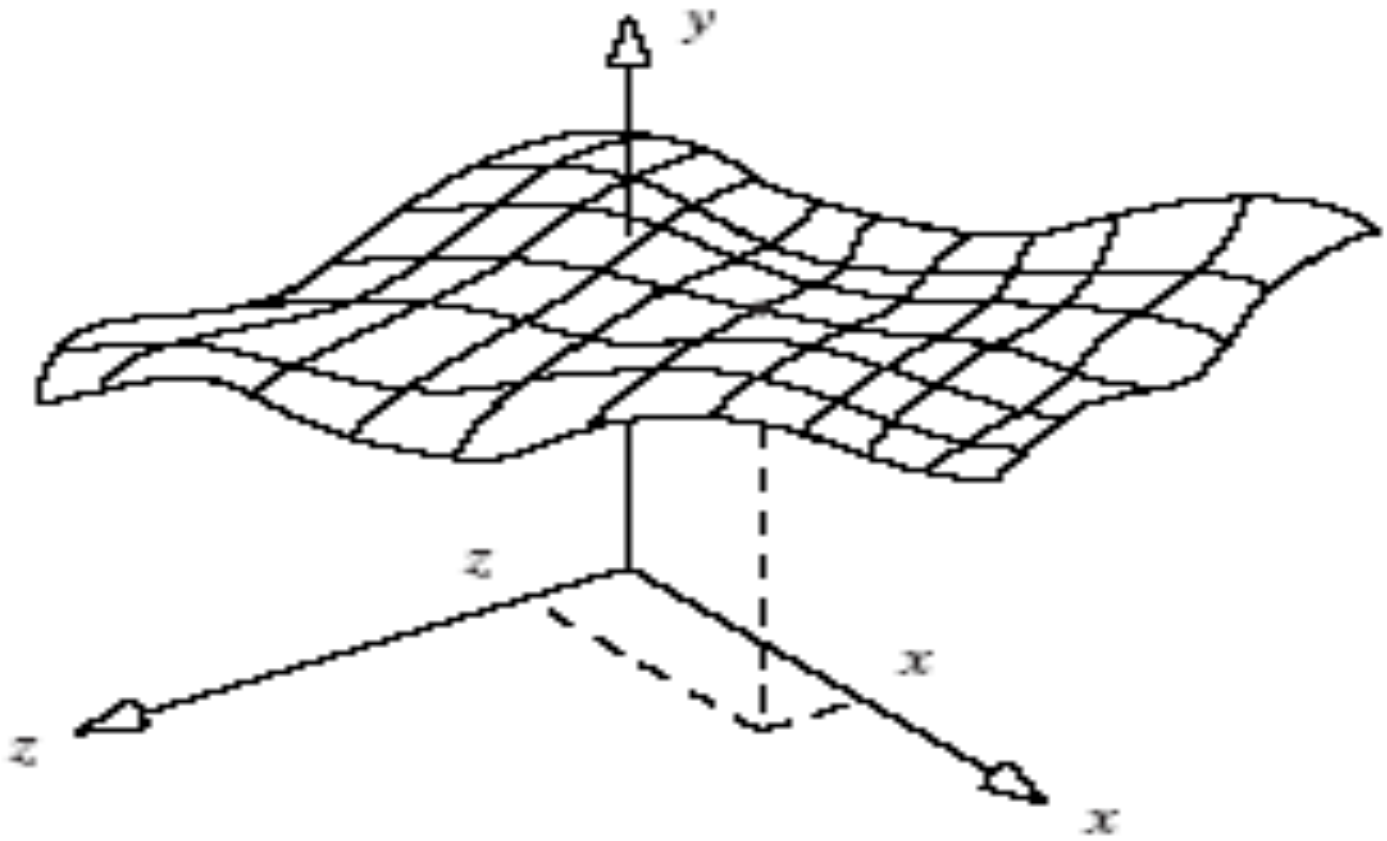
- Physically based systems: the various **objects** in a scene are modeled as connected by springs, gears, electrostatic forces, gravity, or other mechanisms.

Modeling Shapes with Polygonal Meshes

- Generic **Sphere**
- Generic Tapered **Cylinder**
- Generic **Cone**

Surfaces which are *Functions* of Two Variables

Define a single-valued height ***Function*** **y = f(x, z)** of any sort you wish.

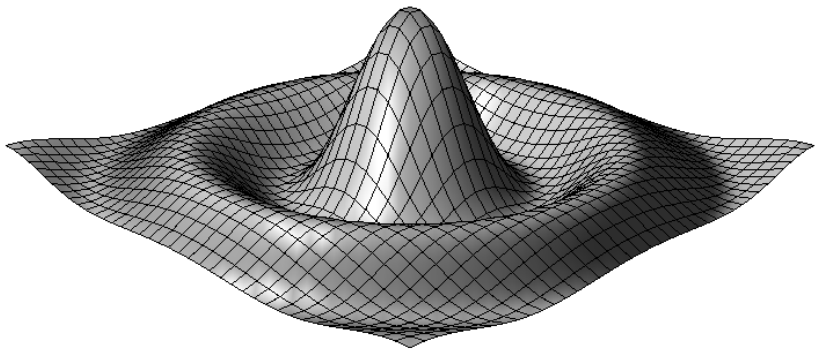


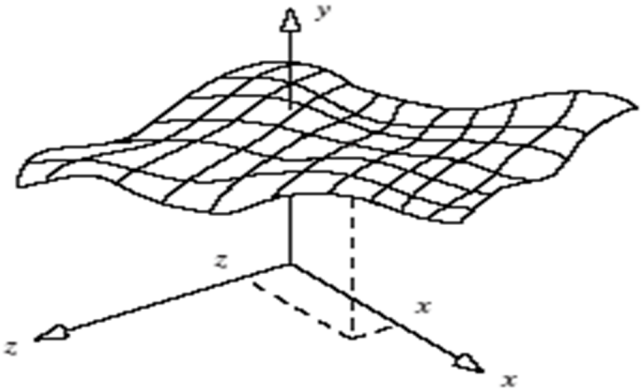
Surfaces which are *Functions* of Two Parameters

Parametric form: P(u, v) = (u, **f**(u, v), v)

Normal vector **n**(u, v) =

Thus *u*-contours lie in planes of constant ***x***, and *v*-contours lie in planes of constant ***z***.





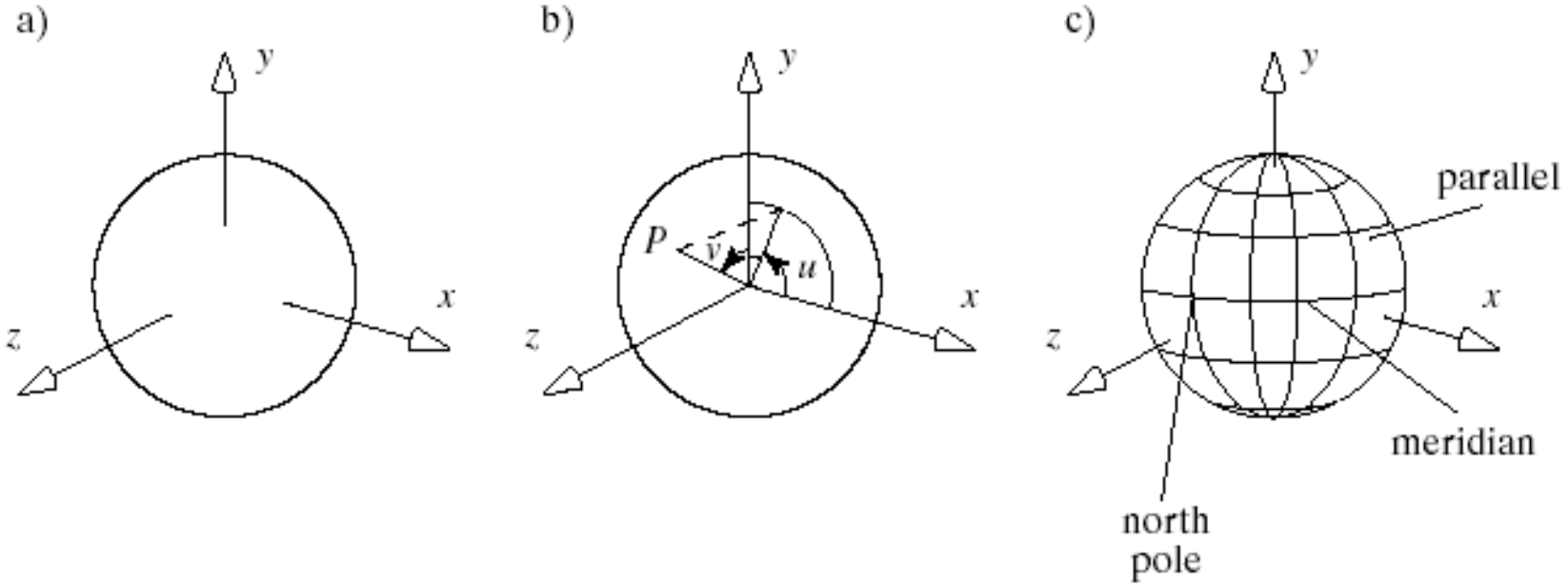
Generic Sphere

Center (0, 0, 0), radius 1;

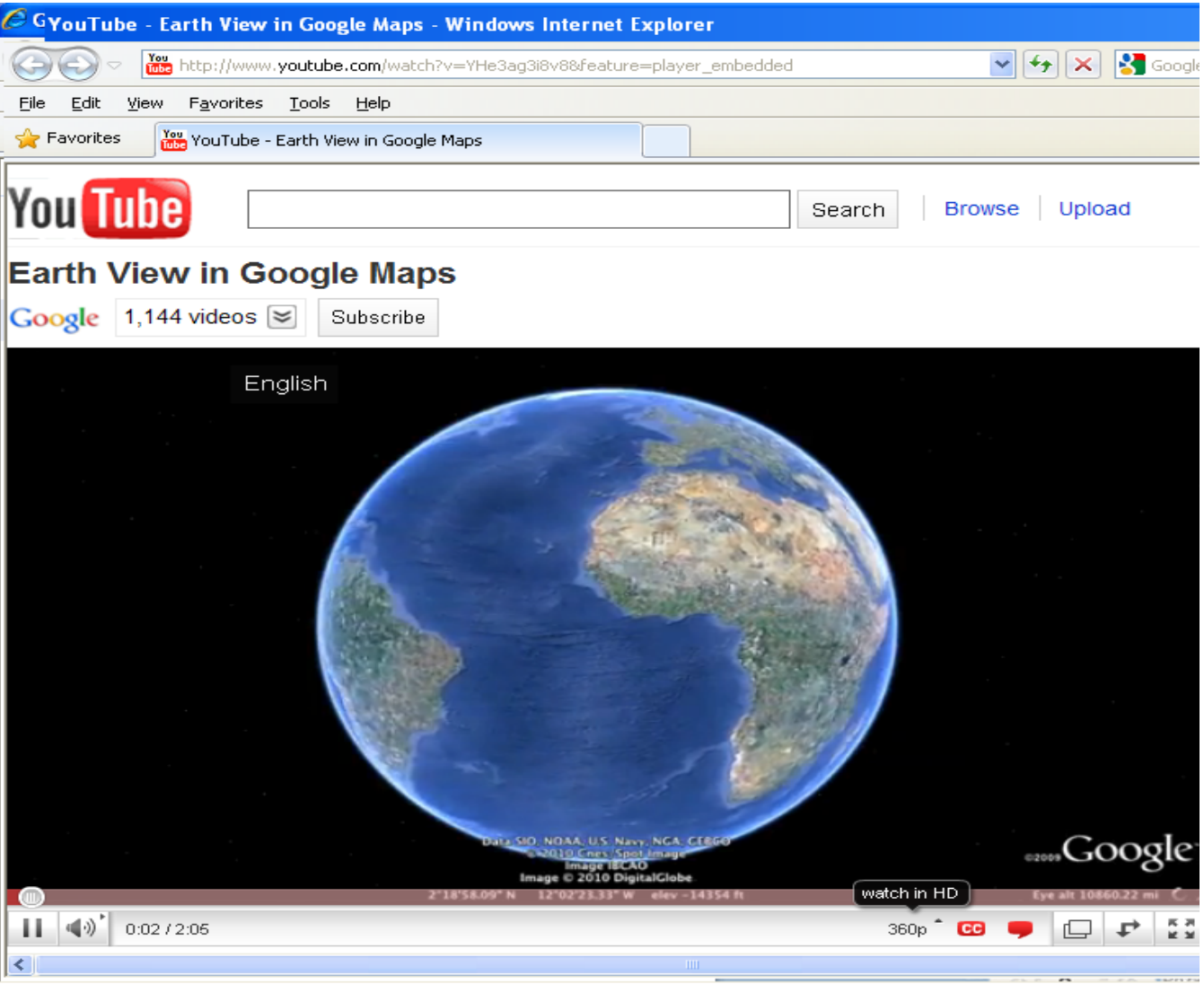
$F(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$, or $F(P)=|P|^2-1$. **Implicit**

$P(u, v) = (\cos v \cos u, \cos v \sin u, \sin v)$, **Parametric**

with $0 \le v \le 2\pi$, $-\pi/2 \le u \le \pi/2$



3D Views in Google Maps



http://www.youtube.com/watch?v=YHe3ag3i8v8&feature=player_embedded

Sphere

$x^2 + y^2 + z^2 - 1$

with $0 \leq v \leq 2\pi$, $-\pi/2 \leq u \leq \pi/2$

u-contours are longitude lines (meridians), azymuth
v-contours are latitude lines (parallels).

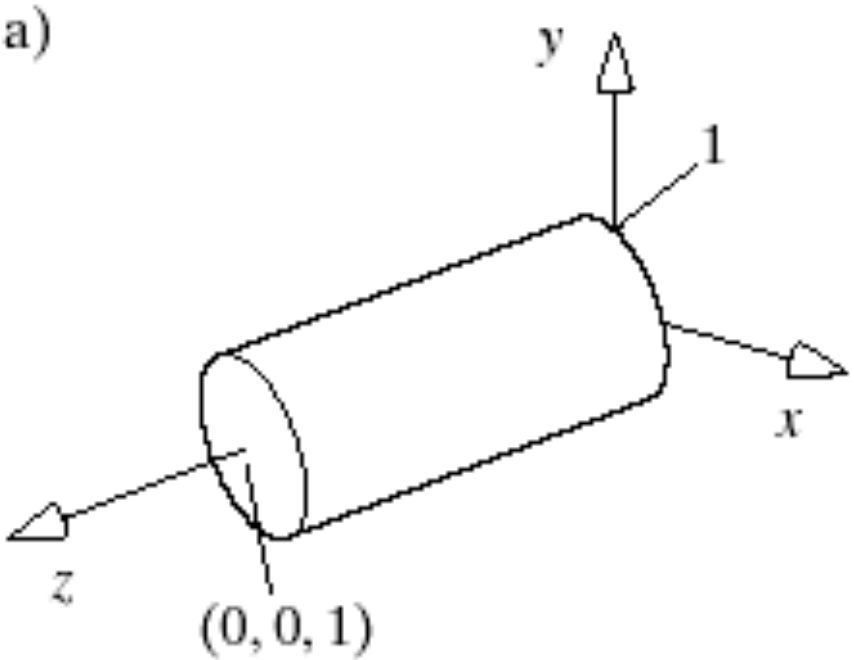
The normal vector **n** (gradient) $2(x, y, z)$ is **radially outward**.

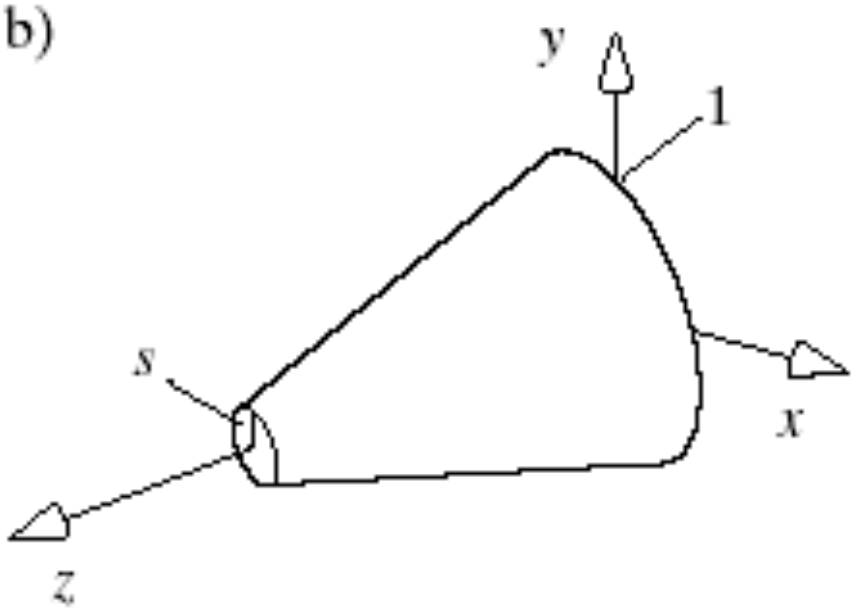
The parametric form is $\mathbf{n}(u, v) = -cos(v)\mathbf{p}(u, v)$, also radially outward. The scale factor $-cos(v)$ will disappear when we normalize **n**.
We must use $\mathbf{p}(u,v)$ rather than $-\mathbf{p}(u,v)$ for the normal **n**, so that it does indeed point radially outward.

Generic Tapered Cylinder

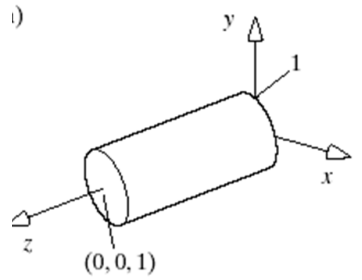
Axis coincides with *z*-axis; circular cross section of radius 1 at base, s when z = 1; extends in *z* from 0 to 1.

The tapered cylinder with an arbitrary value of *s* provides formulas for the generic cylinder and cone by setting *s* to 1 or 0, respectively.





Generic Tapered Cylinder



The wall of the **tapered cylinder** is given by the **implicit form**

$$F(x,y,z) = x^2 + y^2 - (1 + (s-1)z)^2$$

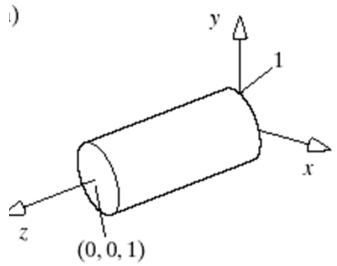
for $0 < z < 1$, and by the **parametric form**

$$P(u,v) = ((1 + (s-1)v)\cos(u), (1 + (s-1)v)\sin(u), v)$$

When the **tapered cylinder** is a solid object, we add two circular discs at its ends: a **base** and a **cap**. The cap is a circular portion of the plane $z = 1$, characterized by the inequality $x^2 + y^2 < s^2$, or given parametrically by

$P(u, v) = (v\ cos(u), v\ sin(u), 1)$ for v in $[0, s]$.

Generic Tapered Cylinder



The **normal vector n** to the wall of the **tapered cylinder** is

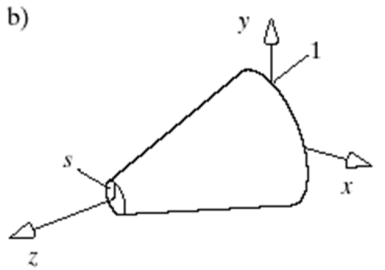
$$\mathbf{n}(x, y, z) = (x, y, -(s - 1)(1 + (s - 1)z)),$$

or in **parametric form**

$$\mathbf{n}(u, v) = (\cos(u), \sin(u), 1 - s).$$

For the generic **cylinder** the **normal** is simply $(\cos(u), \sin(u), 0)$

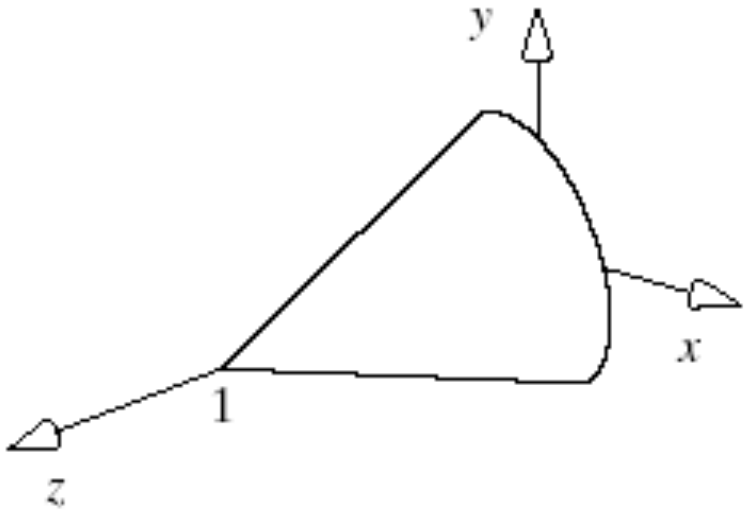
The **normal is directed radially** away from the axis of the cylinder. For the tapered **cylinder it** is also directed radially, but shifted by a constant z-component.



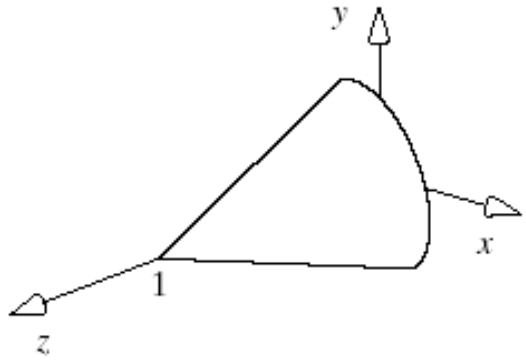
Generic Cone

A **cone** whose axis coincides with the *z*-axis, has a circular cross section of maximum radius 1, and extends in *z* from 0 to 1.

It is a tapered **cylinder** with small radius of $s = 0$.



Generic Cone



Implicit Form

$$F(x, y, z) = x^2 + y^2 - (1 - z)^2 = 0$$

for $0 < z < 1$;

Parametric form

$$P(u, v) = ((1-v)\cos(u), (1- v)\sin(u), v)$$

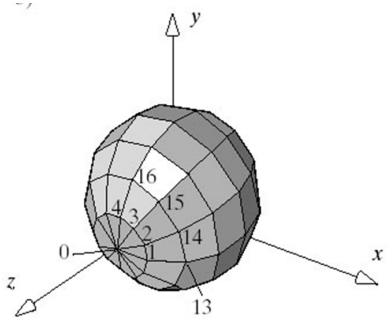
for azimuth u in $[0, 2\pi]$ and v in $[0, 1]$.

Using the results for the tapered **cylinder** again, the **normal vector n** to the wall of the **cone** is $(x, y, 1-z)$

Normal Vectors to the Generic Surfaces

Surface	$\mathbf{n}(u, v)$ at $\mathbf{p}(u, v)$	$\nabla F(x, y, z)$
Sphere	$\mathbf{p}(u, v)$	(x, y, z)
Tapered cylinder	$(\cos(u), \sin(u), 1 - s)$	$(x, y, -(s - 1)(1 + (s - 1)z))$
Cylinder	$(\cos(u), \sin(u), 0)$	$(x, y, 0)$
Cone	$(\cos(u), \sin(u), 1)$	$(x, y, 1 - z)$

Mesh for the Generic Sphere



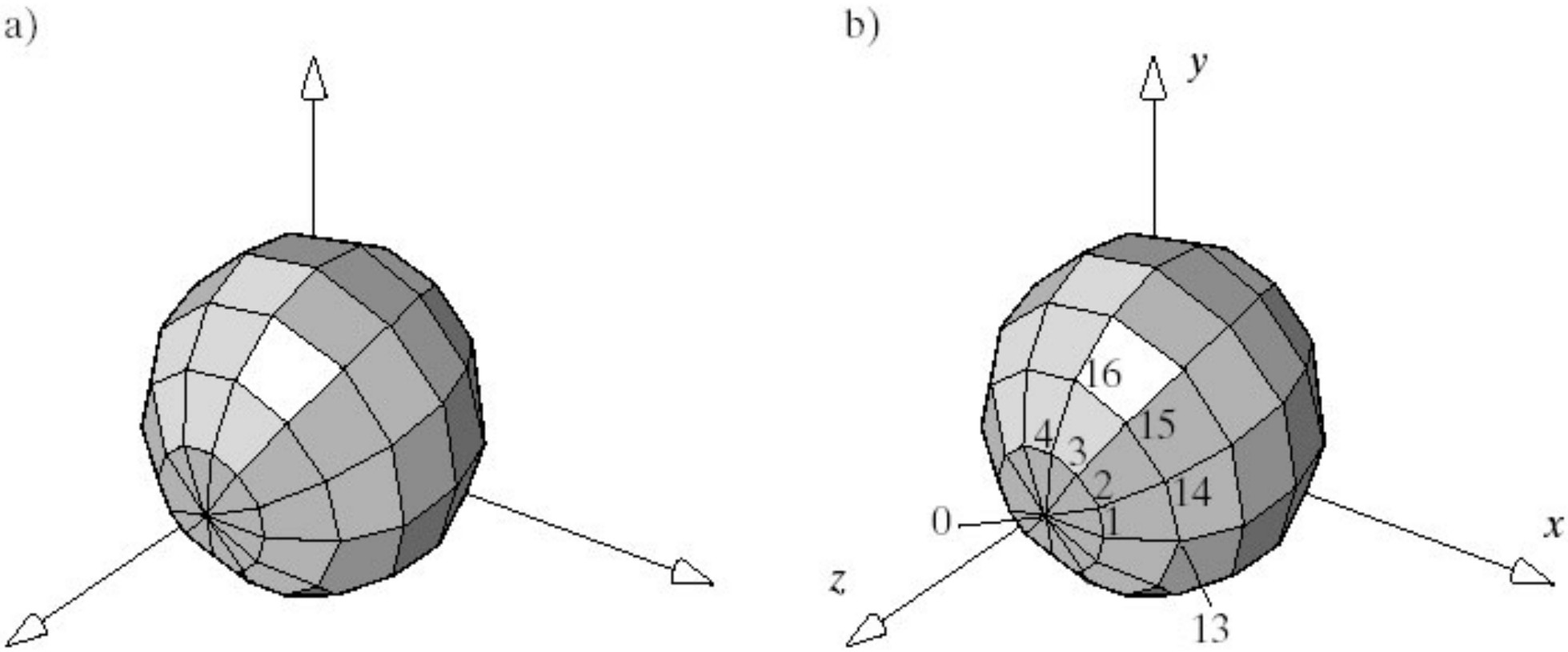
We slice the sphere along azimuth lines and latitude lines.

We slice the sphere into nSlices slices around the equator and nStacks stacks from the South Pole to the North Pole.

The figure shows the example of 10 slices and 8 stacks.

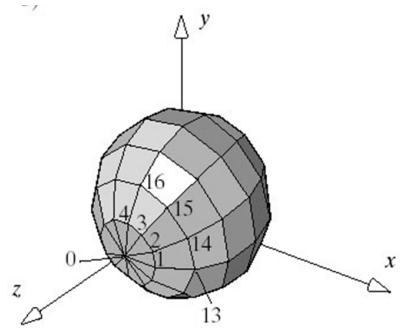
The larger nSlices and nStacks are, the better the mesh approximates a true sphere.

Mesh for the Generic Sphere



10 slices and 8 stacks

Mesh for the Generic Sphere

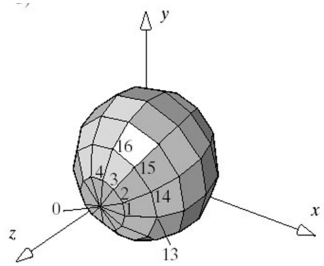


To make **slices** we need **nSlices** values of ***u*** around the equator between 0 and 2π. Usually these are chosen to be **equispaced**: $u_i = 2\pi i/\mathbf{nSlices}$, $i = 0, 1, \ldots, \mathbf{nSlices} - 1$.

We put half of the stacks above the equator and half below. **The top and bottom stacks will consist of triangles; all other faces will be quadrilaterals**.

This requires we define (**nStacks** + 1) values of latitude: $v_j = \pi - \pi\, j/\mathbf{nStacks}$, $j = 0, 1, \ldots, \mathbf{nStacks}$.

Mesh for the Generic Sphere



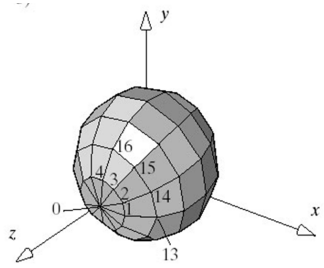
10 slices and 8 stacks

The vertex List: put the north pole in pt[0], the bottom points of the top stack into the next 12 vertices, etc. There will be 98 points.

The normal vector List: norm[k] is the normal for the sphere at vertex pt[k] in parametric form; n(u,v) is evaluated at (u,v) used for the points.

For the sphere this is particularly easy since norm[k] is the same as pt[k].

Mesh for the Generic Sphere



The face List: Put the top triangles in the first 12 faces, the 12 quadrilaterals of the next stack down in the next 12 faces, etc.

The first 3 entries in the face list will contain the data

number of vertices:	**3**	**3**	**3 ...**
vertex indices:	**0 1 2**	**0 2 3**	**0 3 4 ...**
normal indices:	**0 1 2**	**0 2 3**	**0 3 4 ...**

General Meshes

Ultimately, we need a method, such as **makeSurfaceMesh()**, that generates appropriate **meshes** for a given surface $P(u, v)$.

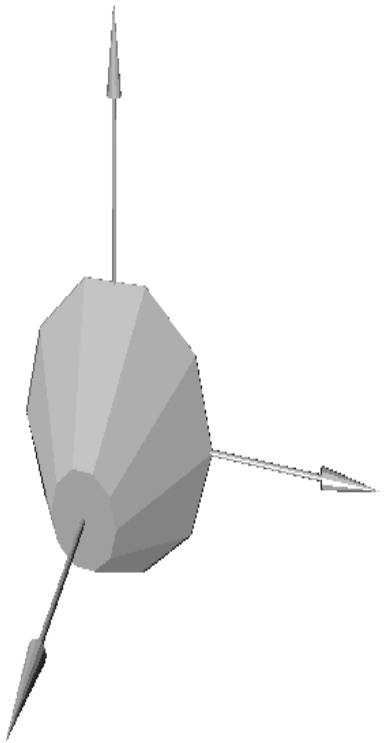
Some graphics packages have routines that are **highly optimized for triangles**, making **triangular meshes** preferable to quadrilateral ones.

We can use the same **vertices but alter the face list** by **replacing each quadrilateral with two triangles**.

For instance, a face that uses vertices 2, 3, 15, 14 might be subdivided into **two triangles**, one using 2, 3, 15 and the other using 2, 15, 14.

Mesh for the Tapered Cylinder

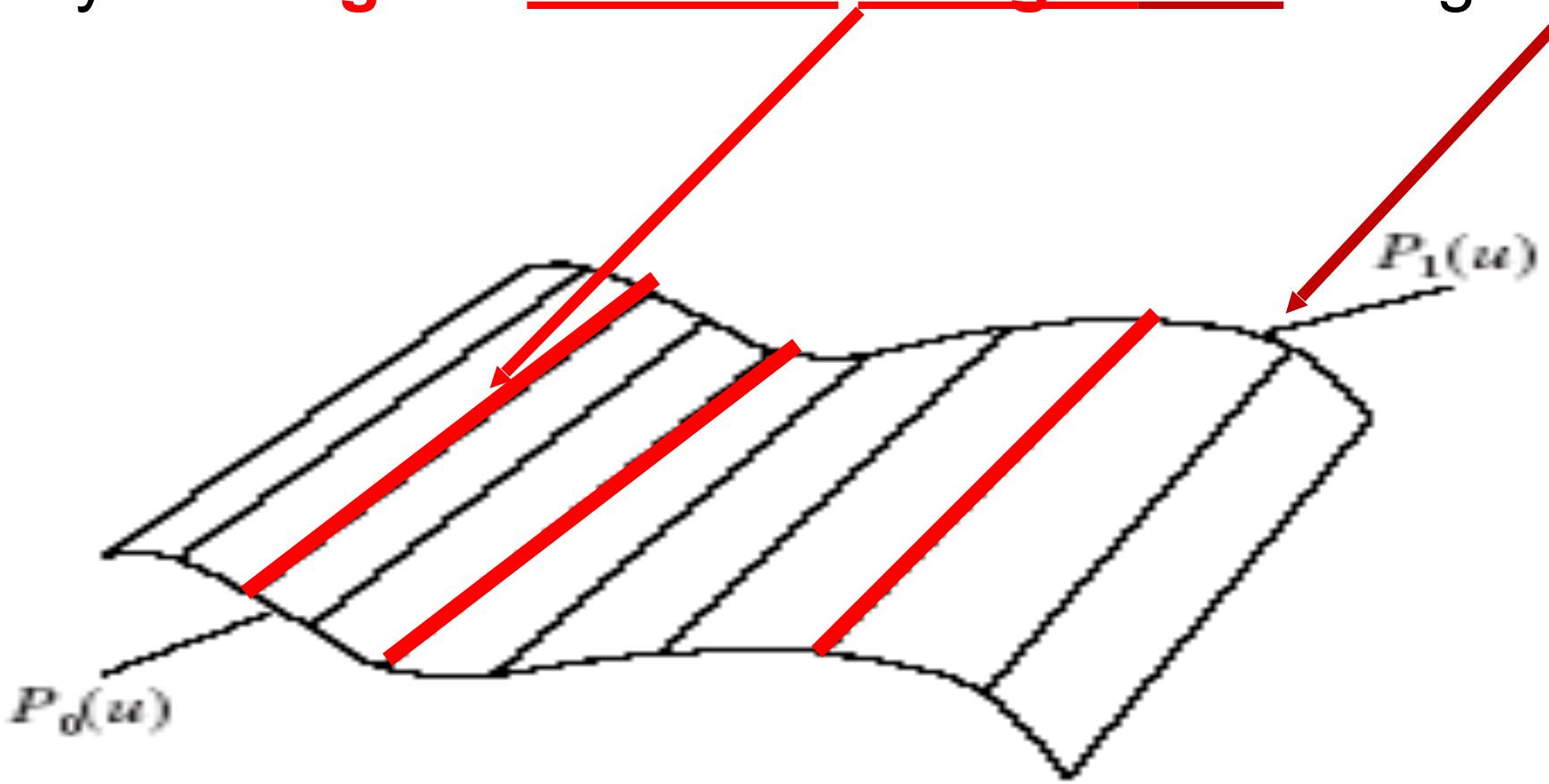
- We use **nSlices** = 10 and **nStacks** = 1.
- A decagon (10) is used for the cap and base.
- If you **prefer to use only triangles**, the walls, the cap, and the base could be dissected into **triangles**.



Ruled Surfaces

Ruled **Surface**: through every **Point**, there passes at least one straight **Line** lying entirely on the **Surface**.

Made by **moving the ends of a straight Line** along **Curves**



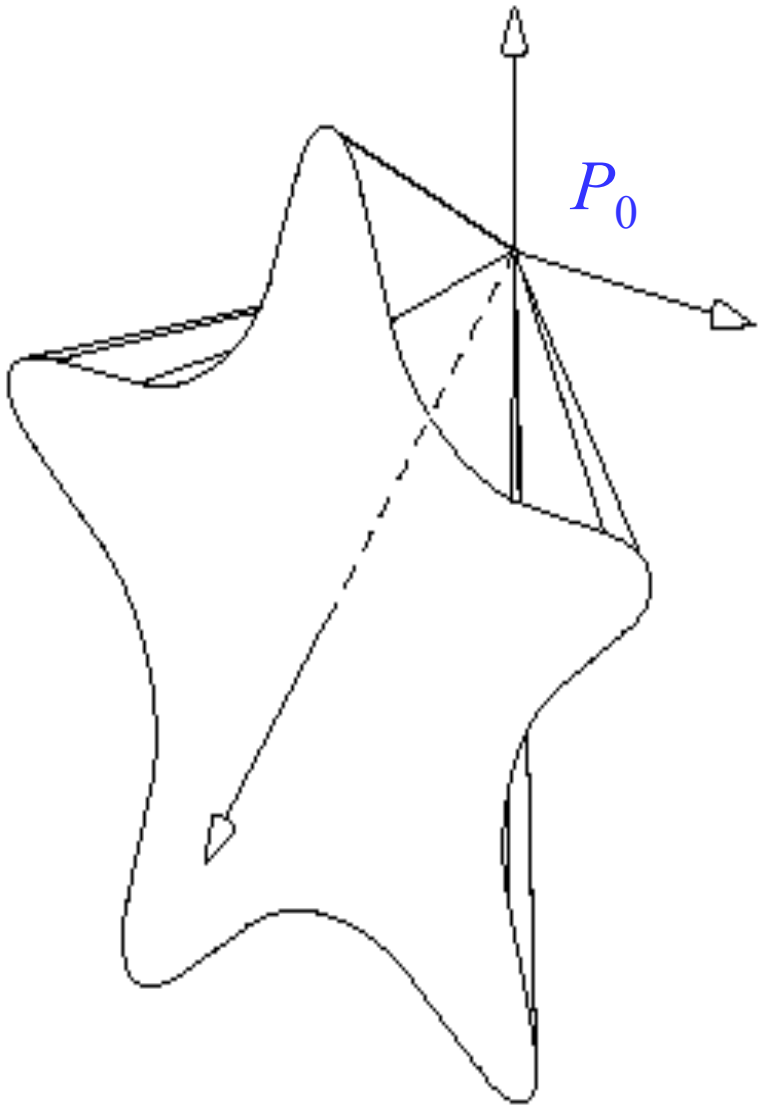
Ruled Surfaces

- A **cone** is a **ruled Surface** for which one of the **Curves**, say,

$P_0(u)$, is a *single* **Point**

$P_0(u) = P_0$, the apex of the **cone**.

$P(u, v) = (1 - v) P_0 + v$

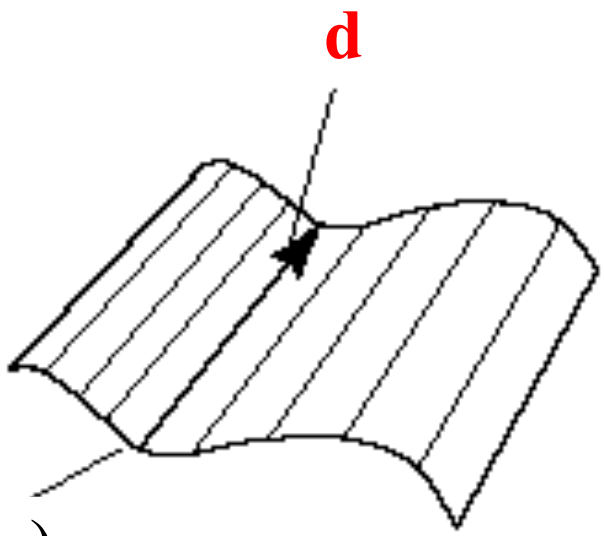


Ruled Surfaces

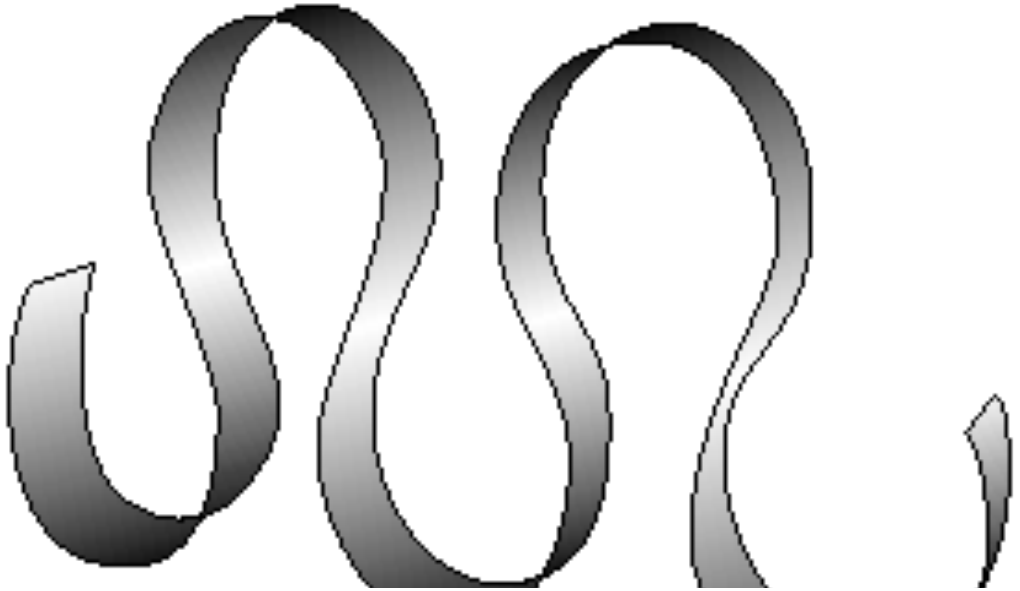
- A **cylinder** is a **ruled Surface** for which $P_1(u)$ is a translated version of $P_0(u)$:

$P_1(u) = P_0(u) + \mathbf{d}$, for some vector **d**.

a)



b)



"**ribbon candy cylinder**", where $P_0(u)$ undulates back and forth like a piece of ribbon

Ruled Surfaces

- The general **cylinder** has the **parametric form** $P(u, v) = P_0(u) + \mathbf{d}v$.

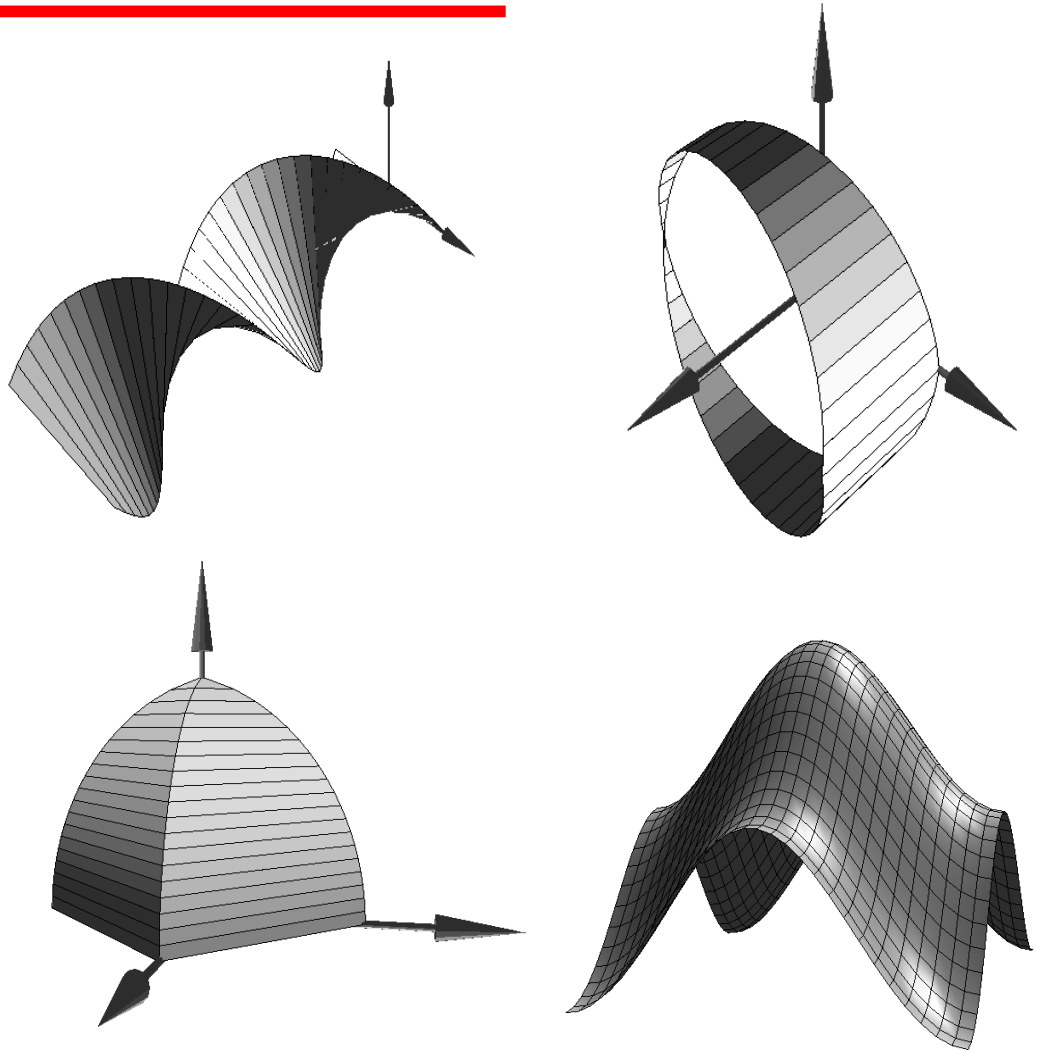
- To be a **true cylinder**, the **Curve** $P_0(u)$ is confined to lie in a **Plane**.

  If $P_0(u)$ is a **circle** the **cylinder** is a **circular cylinder**.

  The direction **d** need not be perpendicular to this **Plane**, but if it is, the **Surface** is called a **right cylinde**

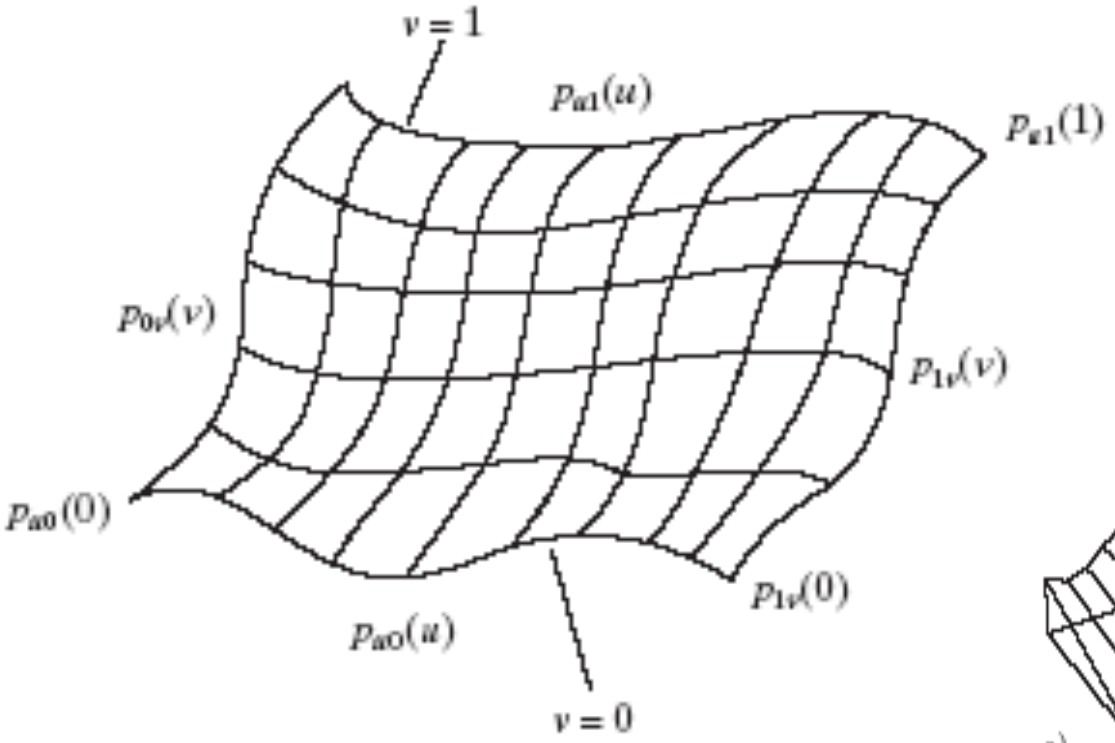
Ruled Surfaces

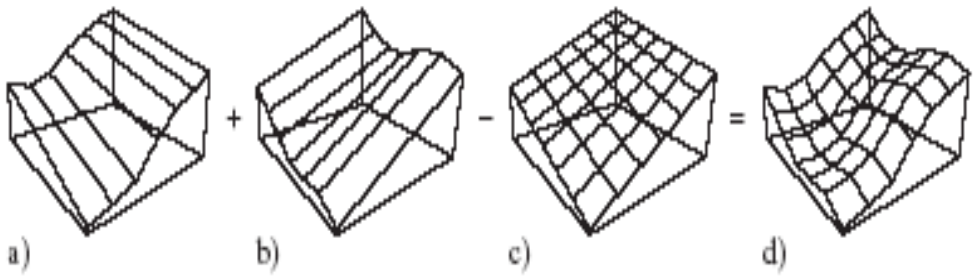
- **Double helix**: $P_0(u)$ and $P_1(u)$ are both helixes that wind around each other.
- **Möbius strip** (has only one edge).
- **Vaulted roof** made up of four ruled surfaces.
- **Coons patch** named after the legendary graphicist Steven Coons.



Coons Patches

- Interpolates 4 boundary **Curves**.
- $P(u, v) = [p_{0v}(v)(1-u) + p_{1v}(v)u] + [p_{u0}(u)(1-v) + p_{u1}(u)v] - [(1-u)(1-v)p_{0v}(0) + u(1-v)p_{1v}(0)] + [v(1-u)p_{0v}(1) + uv\, p_{1v}(1)]$





Surfaces of Revolution

Produced by rotational sweep of profile **Curve** C around an axis.

Curve C(v) = (X(v), Z(v)) is revolved, **generally around the z axis.**

u is the angle of rotation, and v determines the **shape of the Curve**.

When **Point** (X(v), 0, Z(v)) is rotated by angle u, it becomes ((X(v)cos(u), X(v)sin(u), Z(v)).

$$P(u, v) = (X(v)\cos(u), X(v)\sin(u), Z(v))$$

Surfaces of Revolution

- The different positions of the **curve *C*** around the axis are called **meridians**.

- Sweeping **C** completely around generates a full **Circle**, so contours of constant ***v*** are **Circles**, called **parallels**.

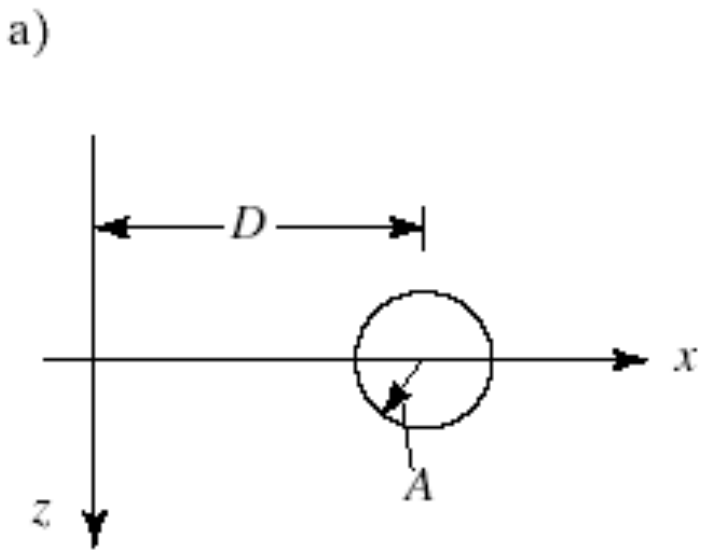
- The **normal vector is**

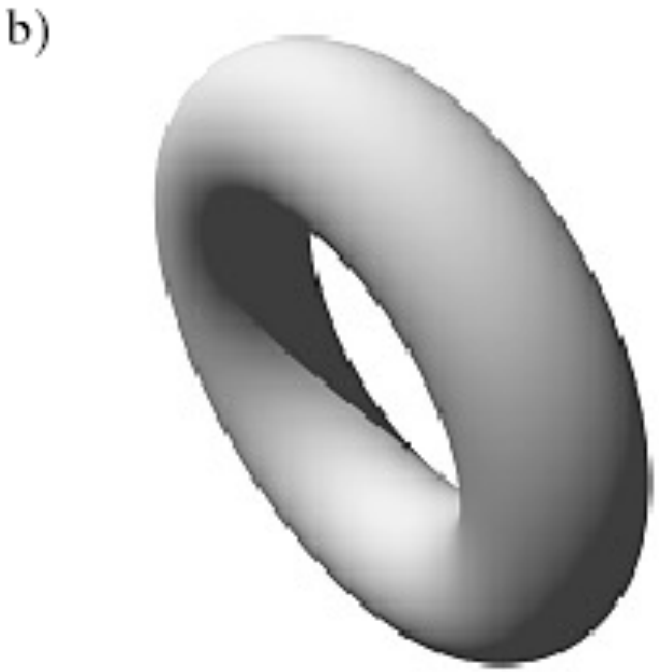
$$\mathbf{n}\,(u, v) = X(v)\,[\dot{Z}(v)\cos(u),\ \dot{Z}(v)\sin(u),\ -\dot{X}(v)].$$

Example

- The **torus** is generated by **sweeping a Circle** displaced by a **distance** D along the x-axis **about the *z*-axis**.

- **The Circle has radius *A*,** so its profile is
  C(**v**) = (*D* + *A* *cos*(**v**), *A* *sin*(**v**)).
- The **torus** has representation
  P(**u**, **v**) = ((*D* + *A* *cos*(**v**)) *cos*(**u**), (*D* + *A* *cos*(*v*)) *sin*(**u**), *A* *sin*(**v**))





Surfaces of Revolution

- A **mesh for a Surface of revolution** is built in a program in the usual way.

We choose a set of ***u*** and ***v*** values, $\{u_i\}$ and $\{v_j\}$, and compute a **vertex at each** from $P(u_i, v_j)$, and a **normal direction** from **n**(u_i, v_j).

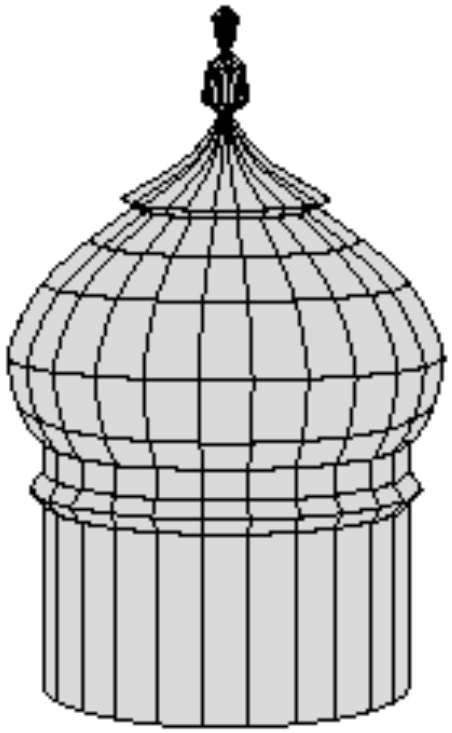
- **Polygonal faces** are built by **joining four adjacent vertices** with **straight lines**.

Example

A **model of the dome** of the **Taj Mahal** in Agra,India.

a)

b)



c)

Example

- A **mesh model of the dome** of the **Taj Mahal**.

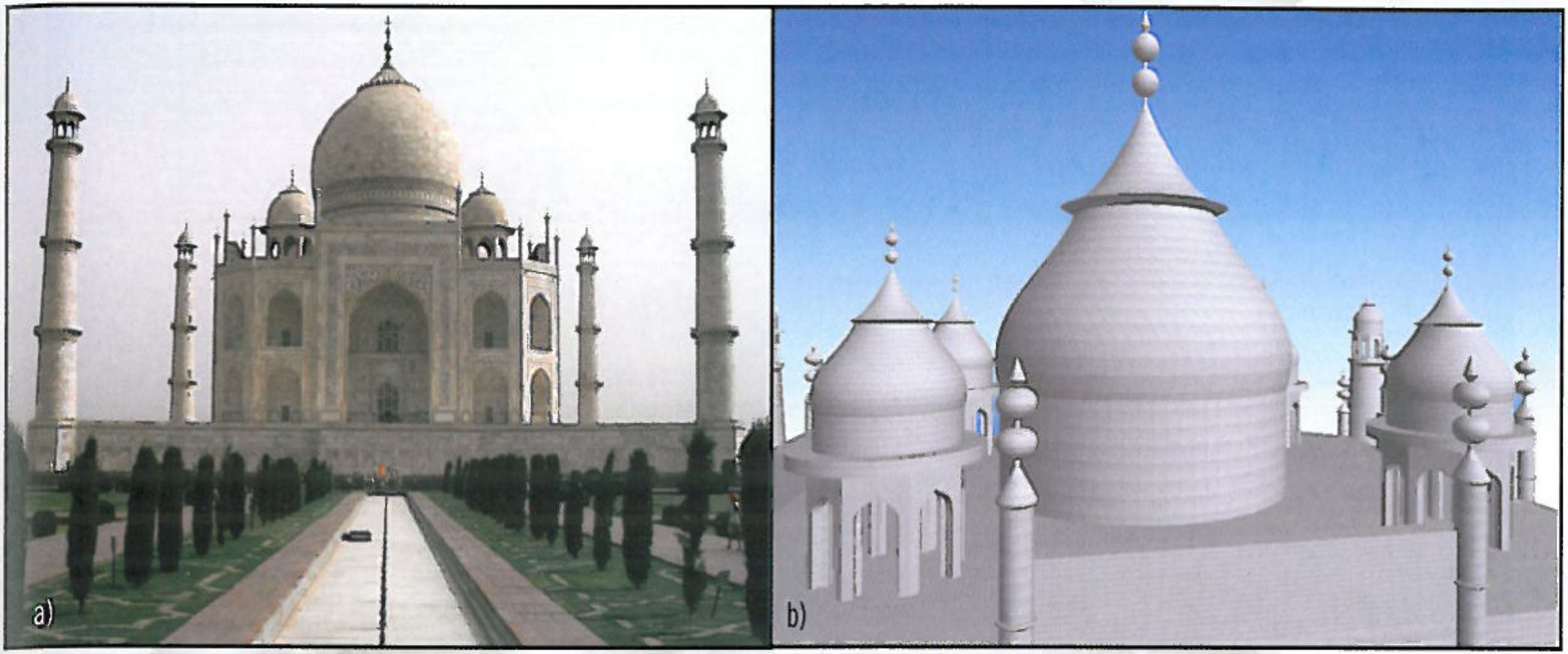
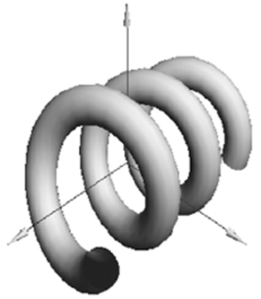


PLATE 13: Part a) An actal photograph of the Taj Mahal. Part b) A screenshot showing mesh model fo the Taj Mahal.

Tubes Based on 3D Curves



We discussed **tubes** based on a "spine" curve $C(t)$ meandering through 3D space.

A polygon was placed at each of a selection of spine points and oriented according to the Frenet frame computed there.

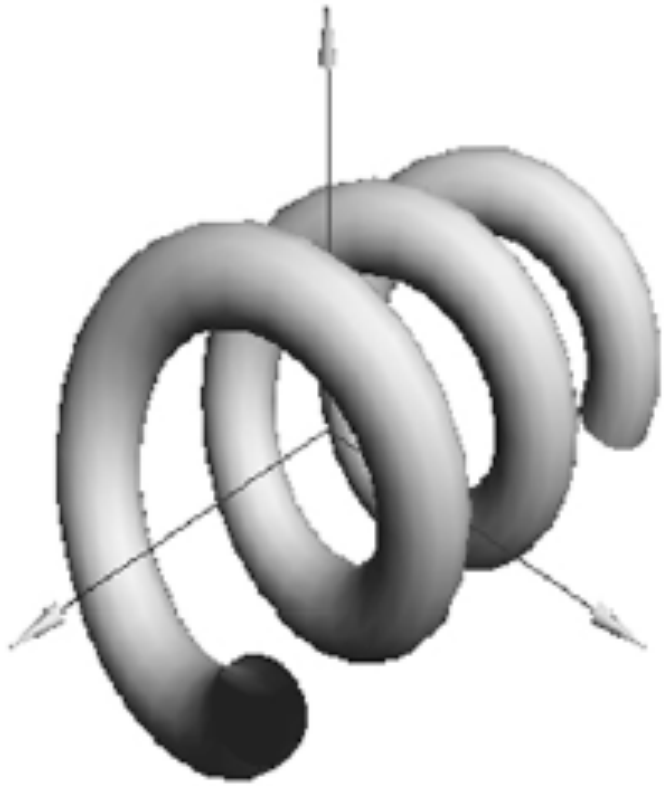
Then **corresponding points on adjacent polygons were connected to form a flat-faced tube** along the spine.

Here we do the same thing, except **we compute the normal to the surface at each vertex** so that **smooth shading** can be performed.

Tubes based on 3-D Curves

- **C(t) is a curve in space** that forms the spine of a **polygon translated** along the **curve**.

- **Circle (cos(u), sin (u), 0)** moves along **helix C**.



Tubes based on 3-D Curves

- The **parametric equation** is

  $$P(u, v) = C(v) + N(v)\cos(u) + B(v)\sin(u),$$

  where **N** and **B** are the **Frenet frame vectors**.

- Then we can build a **mesh** by sampling P(u, v) and **building vertex**, **normal** and **face Lists**.

3D Modeling

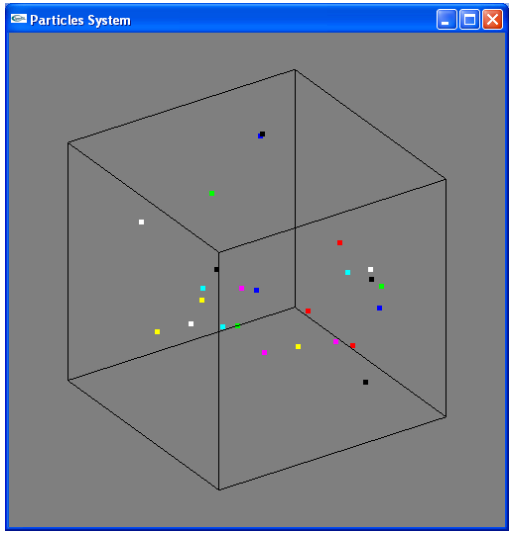
Polygonal meshes capture the shape of complex 3D objects in simple data structures.

- Platonic solids, the Buckyball, geodesic domes, prisms.
- Extruded or swept shapes, and surfaces of revolution.
- Solids with smoothly curved surfaces.

Animated Particle systems: each particle responds to conditions.

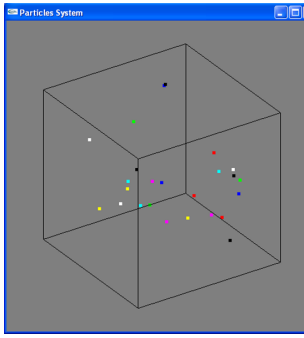
Physically based systems: the various **objects** in a scene are modeled as connected by springs, gears, electrostatic forces, gravity, or other mechanisms.

Particle Systems



Subfield of modeling systems for which there is an attempt to describe in mathematical terms how various objects in a scene behave under the control of forces that act between them.

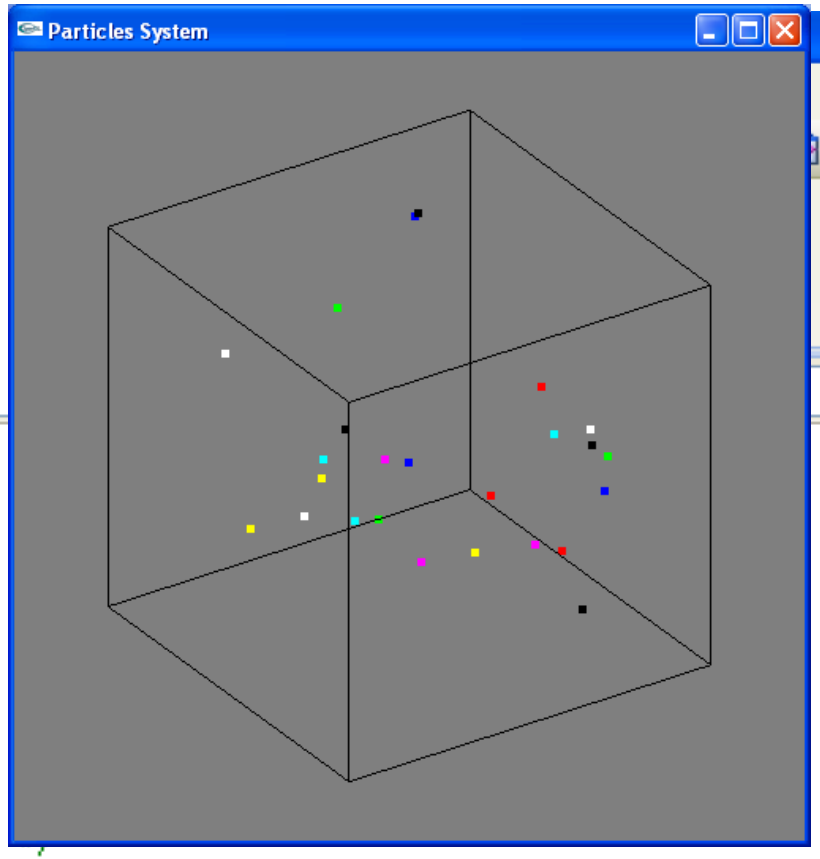
Particle Systems



A **particle system** can keep track of an **enormous number of particles**, each with a position, a velocity, and perhaps a color, lifetime, size, degree of transparency, and shape.

Any of these attributes might be randomly chosen by a **random number generator**, depending on the needs of the application.

particles.c



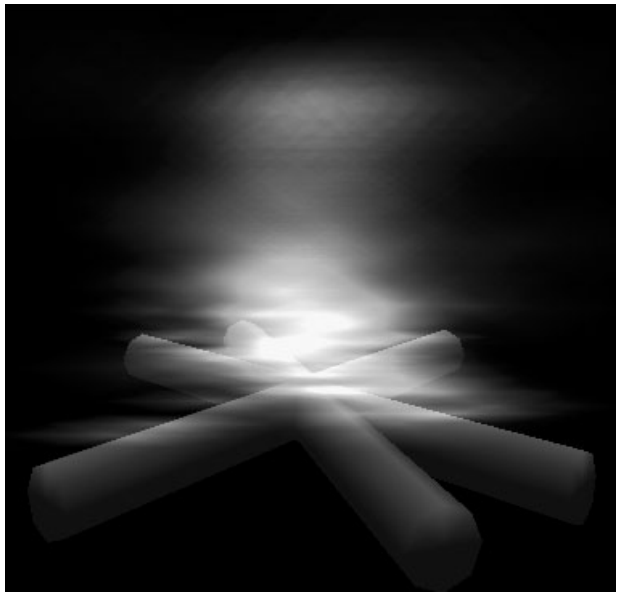
(Unknown Scope)

```
22  void myIdle();
23  void myReshape(int, int);
24  void main_menu(int);
25  void collision(int);
26  float forces(int, int);
27
28  void myinit();
29
30  /* globals */
31
32  int num_particles; /* number of particles
33
34  /* particle struct */
35
3   typedef struct particle
3   {
3       int color;
3       float position[3];
4       float velocity[3];
4       float mass;
4
4
4   } particle;
4
4   particle particles[MAX_NUM_PARTICLES]; /* particle system */
4
```

RUN IT!

Particle Systems
Fire Simulation

- An animation that **simulates fire using a particle system**.
- The central issue in **particle systems** is that **each particle** must be **modeled** with its own set of **parameters**, and **as time moves on**, the **position** and **velocity** of **the particle** must be tracked correctly.
- **Particle systems** require a large amount of memory to **store this information** and a long time **to update it all**.



Particle Systems
Fountain Simulation

- **Particle system** showing water droplets in a fountain.

PLATE 9: A water fountain animation. *(Courtesy of Phillipp Crocoll.) Program available online.*

Physically Based Systems
Flag Simulation

In **physically based modeling**, we describe in mathematical terms how the various forces in a system of **objects** interact to control the motion of **these objects**.

Example: flags in the wind.

Physically Based Systems

- One of the key ingredients is that **different objects collide with one another** and are **possibly deformed in the process**.

- Describing such systems leads to very complex mathematics (ordinary differential equations (**ODE**), partial differential equations (**PDE**), etc.).

- These often must be solved numerically which tends to make the **associated algorithms rather slow**.

Physically Based Systems

In computer games, animation allows the **modeling and viewing of terrain**. The observer can fly over simulated terrain.

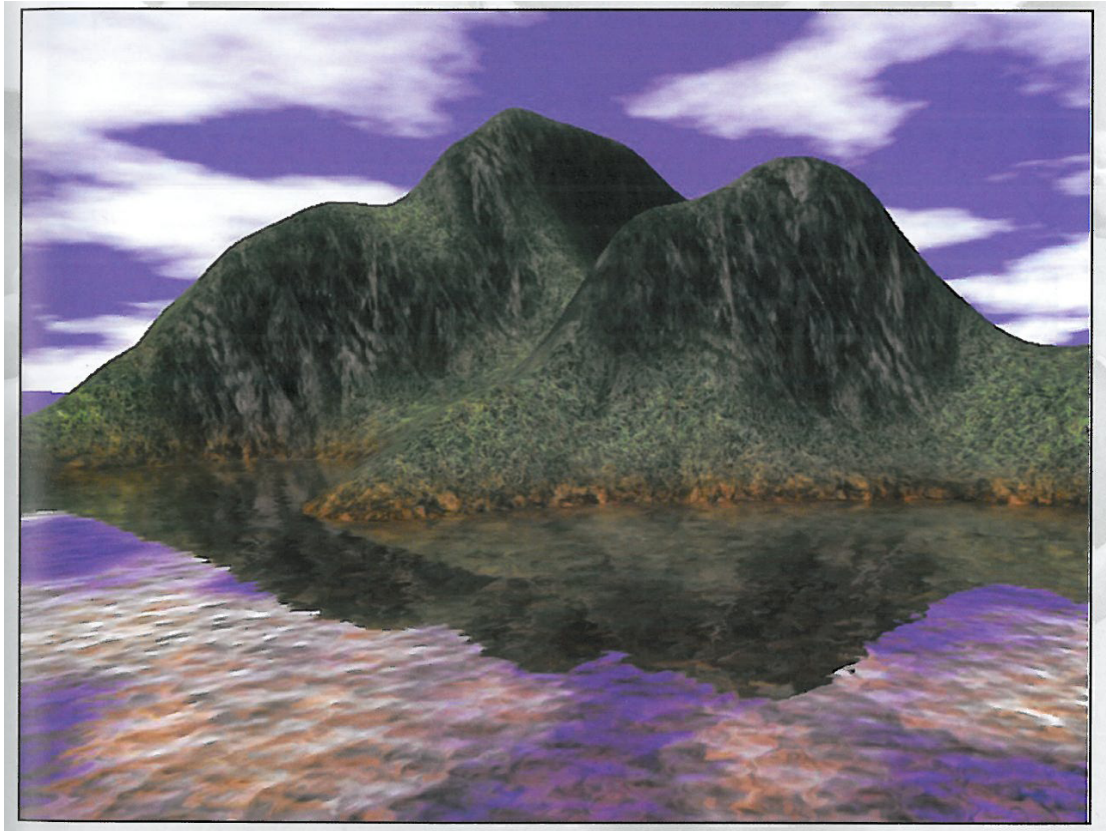


PLATE 14: An image of terrain using mesh models for the mountains and hills and particles for the clouds. *(Courtesy of Rob Hall.)*

Particle Systems Example
Fountain Simulation

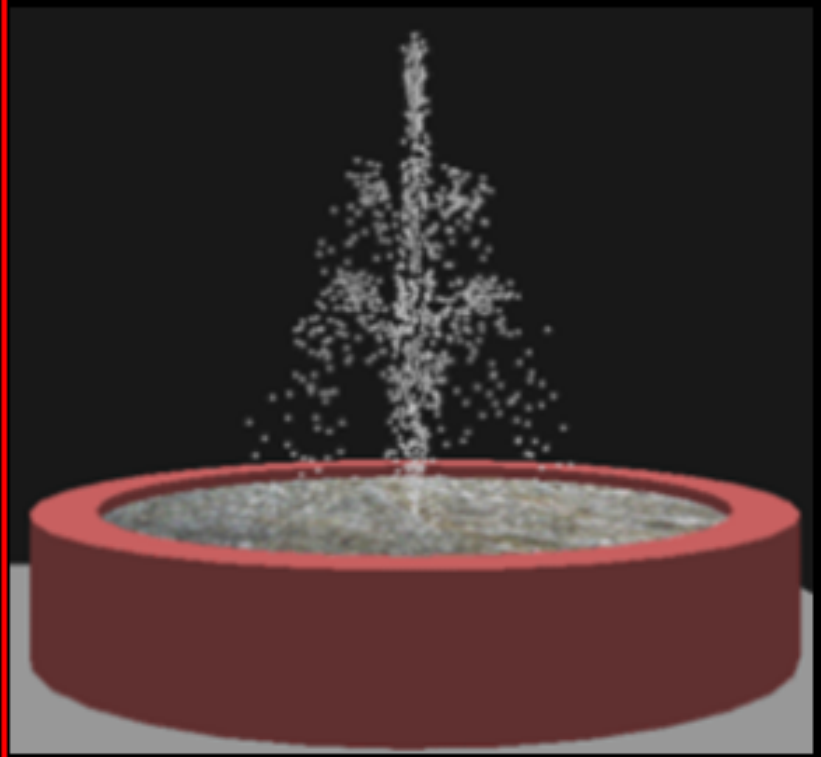
- **Particle system** showing water droplets in a fountain.

- Physical simulation of a drop
- Calculation of the surrounding stone
- Enable blending

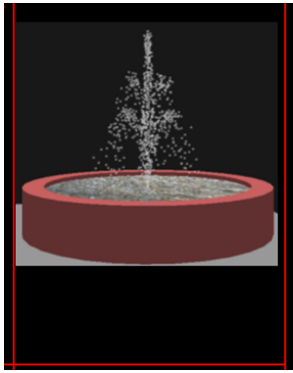
Uses:
- GLUT
- Double buffering
- Depth buffer
- Moving the viewpoint
- User input
- Vertex arrays
- Display Lists
- Texture mapping
- Blending, antialiasing

See also:
Foutain with simulated water

Download:
Complete tutorial (128kb)



Particle Systems Example
Fountain Simulation



Some basic physics...

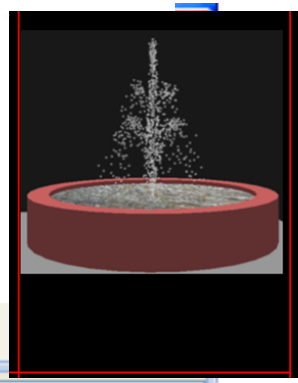
The movement of a drop contains two factors:

- **the direction, how the drop gets out of the fountain and**
- **the gravity**

The **position of a drop** is pretty easy to compute if you know, **how much time has passed since the drop has left the fountain**. You have to **multiply the vector of the constant moving** (how the drop leaves the fountain) **with the time** and **then subtract the squared time multiplied with an acceleration factor.** This **acceleration factor** contains the **weight** of a drop and the power of gravity. This is all.

You now have to know the **direction, how the drop comes** out of the fountain, but this is just a bit calculating with **sine and cosine**.

Particle Systems Example
Fountain Simulation

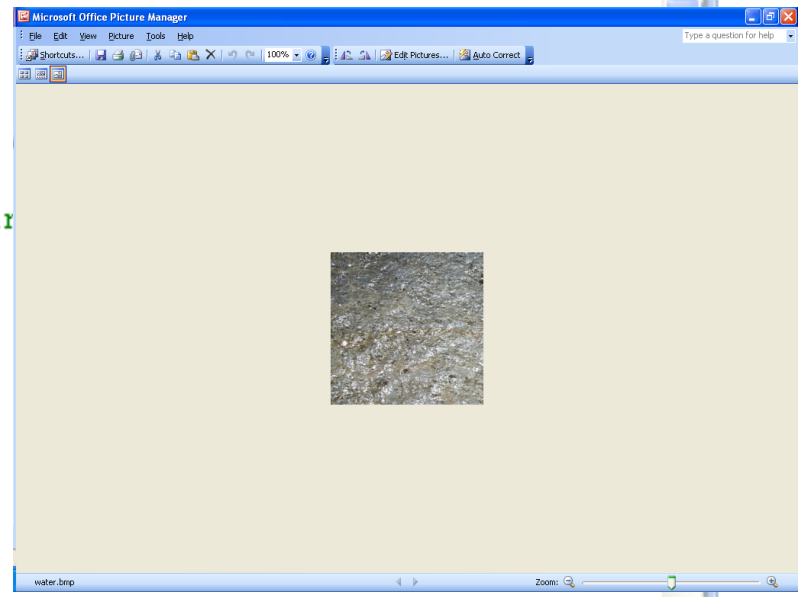


fountain.cpp*

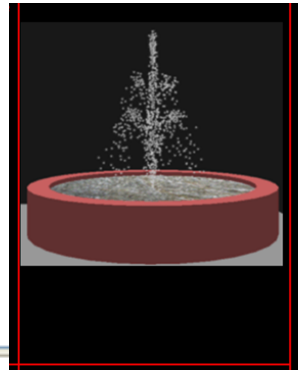
(Unknown Scope)

```
87  class CDrop
88  {
89  private:
90      GLfloat time;           // How many steps the drop was "outside", whe
91                              // falls into the water, time is set back to 
92      SVertex ConstantSpeed;
93      GLfloat AccFactor;
94  public:
95      void SetConstantSpeed (SVertex NewSpeed);
96      void SetAccFactor(GLfloat NewAccFactor);
97      void SetTime(GLfloat NewTime);
98      void GetNewPosition(SVertex * PositionVertex);  //ir
99  };
100
101 void CDrop::SetConstantSpeed(SVertex NewSpeed)
102 {
103     ConstantSpeed = NewSpeed;
104 }
105
106 void CDrop::SetAccFactor (GLfloat NewAccFactor)
107 {
108     AccFactor = NewAccFactor;
109 }
110
111 void CDrop::SetTime(GLfloat NewTime)
112 {
```

water.bmp



Particle Systems Example
Fountain Simulation



(Unknown Scope)

```
111 void CDrop::SetTime(GLfloat NewTime)
112 {
113     time = NewTime;
114 }
115
116 void CDrop::GetNewPosition(SVertex * PositionVertex)
117 {
118     SVertex Position;
119     time += 1.0;
120     Position.x = ConstantSpeed.x * time;
121     Position.y = ConstantSpeed.y * time - AccFactor * time * time;
122     Position.z = ConstantSpeed.z * time;
123     PositionVertex->x = Position.x;
124     PositionVertex->y = Position.y + WaterHeight;
125     PositionVertex->z = Position.z;
126     if (Position.y < 0.0)
127     {
128         /*the drop has fallen into the water. The problem is now, that we canno
129         set time to 0.0, because if there are more "DropsPerRay" than "TimeNeed
130         several drops would be seen as one. Check it out.
131         */
132         time = time - int(time);
133         if (time > 0.0) time -= 1.0;
134     }
135
136 }
```

Introduction

- Most important of procedural methods

- Used to model

  Natural phenomena

  - Clouds
  - Terrain
  - Plants

  Crowd Scenes

  Real **physical processes**

PHYSICALLY- BASED MODELS AND PARTICLE SYSTEMS

- **In fields such as scientific visualization, this flexibility allows mathematicians to see shapes that do not exist in the usual three-dimensional space and to display information in new ways**

- **Recently, researchers have become interested in physically-based modeling, a style of modeling in which the graphical objects obey physical laws. Such modeling can follow either of two related paths.**

  In one, we model the physics of the underlying process and use the physics to drive the graphics.

  The other approach is to use a combination of basic physics and mathematical constraints to control the dynamic behavior of our objects.

PHYSICALLY- BASED MODELS AND PARTICLE SYSTEMS

- **Particle systems have been used to generate a wide variety of behaviors in a number of fields.**

  In fluid dynamics, people use particle systems to model turbulent behavior. Rather than solving partial differential equations, we can simulate the behavior of the system by following a group of particles that is subject to a variety of forces and constraints.

  We can also use particles to model solid objects. For example, a deformable solid can be modeled as a three-dimensional array of particles that are held together by springs. When the object is subjected to external forces, the particles move and their positions approximate the shape of the solid object.

Newtonian Particle

- **Particle system** is a set of particles
- Each particle is an ideal point mass **m**

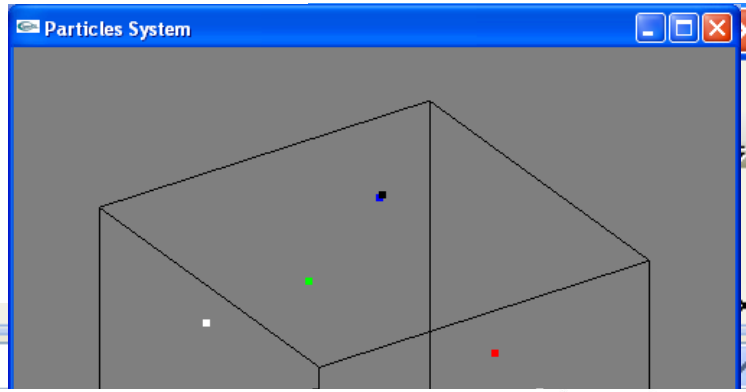
- Six degrees of freedom
  - **Position**
  - **Velocity**

- Each particle obeys **Newtons' law**

$$\mathbf{f = ma}$$

Note: both force **f** and acceleration **a** are **vectors**!

Newtonian Particle



We conclude our development of partial systems by building a simple particle system that can be expanded to more complex behaviors. Our particles are all Newtonian so their state is described by their positions and velocities. In addition, each particle can have its own color and mass. We start with the following structure:

```
typedef struct particle
{
     int color;
     float position[3];
     float velocity[3];
     float mass;
} particle;
```

A particle system is an array of particles:

```
particle particles[MAX_NUM_PARTICLES];
```

particles.c

```
33
34  int num_particles; /* number of particl
35
36  /* particle struct */
37
38  typedef struct particle
39  {
40        int color;
41        float position[3];
42        float velocity[3];
43        float mass;
44
45
46  } particle;
47
48  particle particles[MAX_NUM_PARTICLES];
49
50  /* initial state of particle system */
51
52  int present_time;
53  int last_time;
54  int num_particles = INITIAL_NUM_PARTICLES;
55  float point_size = INITIAL_POINT_SIZE;
56  float speed = INITIAL_SPEED;
57  bool gravity = FALSE; /* gravity off */
58  bool elastic = FALSE; /* restitution off */
```

Displaying Particle

Given the position of a particle, we can display it using any set of primitives, either geometric or raster, that we like. A simple starting point would be to display each particle as a point. Here is a display callback that loops through the array of particles:

```
void myDisplay()
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
    for(i=0; i<num_particles; i++)

       glColor3fv(colors[particles[i].color]);
       glVertex3fv(particles[i].position);

    glEnd();
}
```

The colors are stored in an array `colors` as follows:

particles.c

```
247
248 void myDisplay()
249 {
250     int i;
251
252     glClear(GL_COLOR_BUFFER_BIT);
253     glBegin(GL_POINTS); /* render all particles */
254     for(i=0; i<num_particles; i++)
255     {
256        glColor3fv(colors[particles[i].color]);
257        glVertex3fv(particles[i].position);
258     }
259     glEnd();
260     glColor3f(0.0,0.0,0.0);
261     glutWireCube(2.2);  /* outline of box */
262     glutSwapBuffers();
263 }
```

Value | Type

Output

'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11
'Lecture 11

Particle Equations

Position vector for the particle

$$\mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix},$$

Velocity Vector for the particle

$$\mathbf{v}_i = \begin{bmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{z}_i \end{bmatrix} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{bmatrix}.$$

Newton's Second Law

$$\dot{\mathbf{p}}_i = \mathbf{v}_i,$$

Hard part is defining force vector

$$\dot{\mathbf{v}}_i = \frac{1}{m_i}\mathbf{f}_i(t).$$

$$\mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix},$$

Particle Equations

$$\mathbf{v}_i = \begin{bmatrix} \dot{x}_i \\ \dot{y}_i \\ \dot{z}_i \end{bmatrix} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{bmatrix}.$$

- **Hence, the dynamics of a system of n particles is governed by a set of 6n coupled, ordinary differential equations (ODE).**

- **In addition to its state, each particle may have a number of attributes, including its mass (m_i), and a set of properties that can alter what its behavior is and how it is displayed.**

$$\dot{\mathbf{p}}_i = \mathbf{v}_i,$$

$$\dot{\mathbf{v}}_i = \frac{1}{m_i}\mathbf{f}_i(t).$$

- **Each particle may represent a person in a crowd scene, or a molecule in a chemical-synthesis application, or a piece of cloth in the simulation of a flag blowing in the wind.**

- **In each case, the underlying particle system governs the location and the velocity of the center of mass of the particle. Once we have the location of a particle, we can place the desired graphical object at this location.**

$$\mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix},$$

Particle Equations

$$\dot{\mathbf{p}}_i = \mathbf{v}_i,$$

$$\dot{\mathbf{v}}_i = \frac{1}{m_i}\mathbf{f}_i(t).$$

The forces on the particles, $\{f_i\}$, determines the behavior of the system.

These forces are based on the state of the particle and change with time.

We can base these forces on

- **simple physical principles, such as spring forces, or**
- **on physical constraints that we wish to impose on the system; or**
- **we can base them on external forces, such as gravity, that we wish to apply to the system.**

Designing the forces carefully, we can obtain the desired system behavior.

Solution of Particle Systems

```
float time, delta, state[6n], force[3n];
state = initial_state();
for (time = t0; time<final_time,
 time+=delta)
{
   /* compute force */
      force =  force_function(state, time);

      /* apply standard differential equation solver */
      state = ode(force, state, time,
 delta);

   /* display result */
      render(state, time)
}
```

Function force_function that computes the forces on each particle???

Updating Particle Positions

```
void myIdle()
{
    int i, j, k;
    float dt;
    present_time = glutGet(GLUT_ELAPSED_TIME);
    dt = 0.001*(present_time -  last_time);
    for(i=0; i<num_particles; i++)
    {
       for(j=0; j<3; j++)
       {
           particles[i].position[j]+=dt*particles[i].velocity[j];
           particles[i].velocity[j]+=dt*forces(i,j)/particles[i].mass;
       }
        collision(i);
    }
    if(repulsion) for(i=0;i<num_particles;i++) for(k=0;k<i;k++)
    {
            d2[i][k] = 0.0;
            for(j=0;j<3;j++) d2[i][k]+= (particles[i].position[j]-
              particles[k].position[j])*(particles[i].position[j]-
              particles[k].position[j]);
            d2[k][i]=d2[i][k];
    }
    last_time = present_time;
    glutPostRedisplay();
}
```

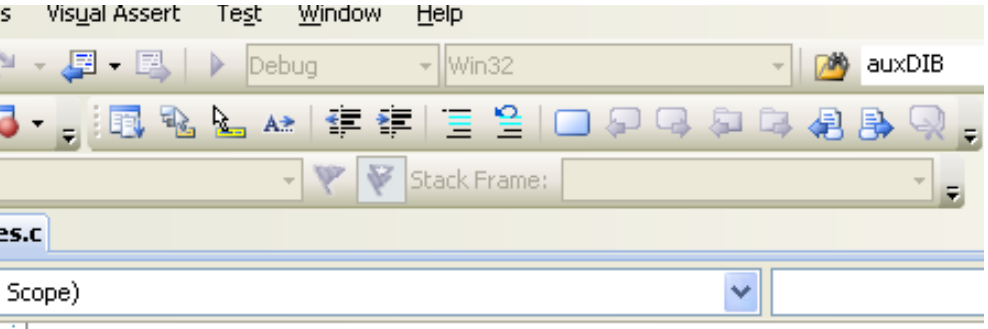
```
float last_time, present_time;
float num_particles;

void myIdle()
{
    int i, j;
    float dt;
    present_time = glutGet(GLUT_ELAPSED_TIME); /* in milliseconds */
    dt = 0.001*(present_time -  last_time); /* in seconds */
    for(i=0; i<num_particles; i++)
    {
       for(j=0; j<3; j++)
       {
           particles[i].position[j]+=dt*particles[i].velocity[j];
           particles[i].velocity[j]+=dt*forces(i,j)/particles[i].mass;
       }
        collision(i);
    }
    last_time = present_time;
    glutPostRedisplay();
}
```

Initialization

```
float num_particles = INITIAL_NUM_PARTICLES;
float speed = INITIAL_SPEED;

void myinit()
{
     int  i, j;
     for(i=0; i<num_particles; i++)
     {
          particles[i].mass = 1.0; /* default particle mass */
          particles[i].color = i%8; /* 8 colors in color array */
          for(j=0; j<3; j++)
          {
               particles[i].position[j] = 2.0*((float) random()/
                  RAND_MAX)-1.0;
               particles[i].velocity[j] = speed*2.0*((float) random()/
                  RAND_MAX)-1.0;
          }
     }
}
```




```
}
void myinit()
{
        int  i, j;

 /* set up particles with random locations and velocities */

        for(i=0; i<num_particles; i++)
        {
             particles[i].mass = 1.0;
             particles[i].color = i%8;
             for(j=0; j<3; j++)
             {
                  particles[i].position[j] = 2.0*((float) rand()/RAND_MAX)-1.0;
                  particles[i].velocity[j] = speed*2.0*((float) rand()/RAND_MAX)-1.0;
             }
        }
        glPointSize(point_size);


/* set clear color to grey */

        glClearColor(0.5, 0.5, 0.5, 1.0);
}
```

Collisions

We use the collision function to keep the particles inside the initial axis-aligned box. Our strategy is to increment the position of each particle and then check to see if the particle has crossed one of the sides of the box. If it has crossed a side then we can treat the bounce as a reflection. Thus, we need only to change the sign of the velocity in the normal direction and place the particle on the other side of the box. If the coefficient of restitution is less than 1.0, the particles will slow down when they hit a side of the box:

```
float coef; /* coefficient of restitution */

void collision(int n)
{
   int i;
   for (i=0; i<3; i++)
   {
      if(particles[n].position[i]>=1.0)
      {
         particles[n].velocity[i] = -coef*particles[n].velocity[i];
         particles[n].position[i] =
                                    1.0-coef*(particles[n].position[i]-1.0);
      }
      if(particles[n].position[i]<=-1.0)
      {
         particles[n].velocity[i] = -coef*particles[n].velocity[i];

         particles[n].position[i] = -1.0-
                                    coef*(particles[n].position[i]+1.0);
      }
   }
}
```

Visual Assert Test Window Help

Debug Win32 auxDIB

Stack Frame:

```
}
```

```
void collision(int n)

/* tests for collisions against cube and reflect particles

{
     int i;
     for (i=0; i<3; i++)
     {
          if(particles[n].position[i]>=1.0)
          {
               particles[n].velocity[i] = -coef*particles[n].velocity[i];
               particles[n].position[i] = 1.0-coef*(particles[n].position[i]-1.0);
          }
          if(particles[n].position[i]<=-1.0)
          {
               particles[n].velocity[i] = -coef*particles[n].velocity[i];
               particles[n].position[i] = -1.0-coef*(particles[n].position[i]+1.0);
          }
     }
}
```

Output

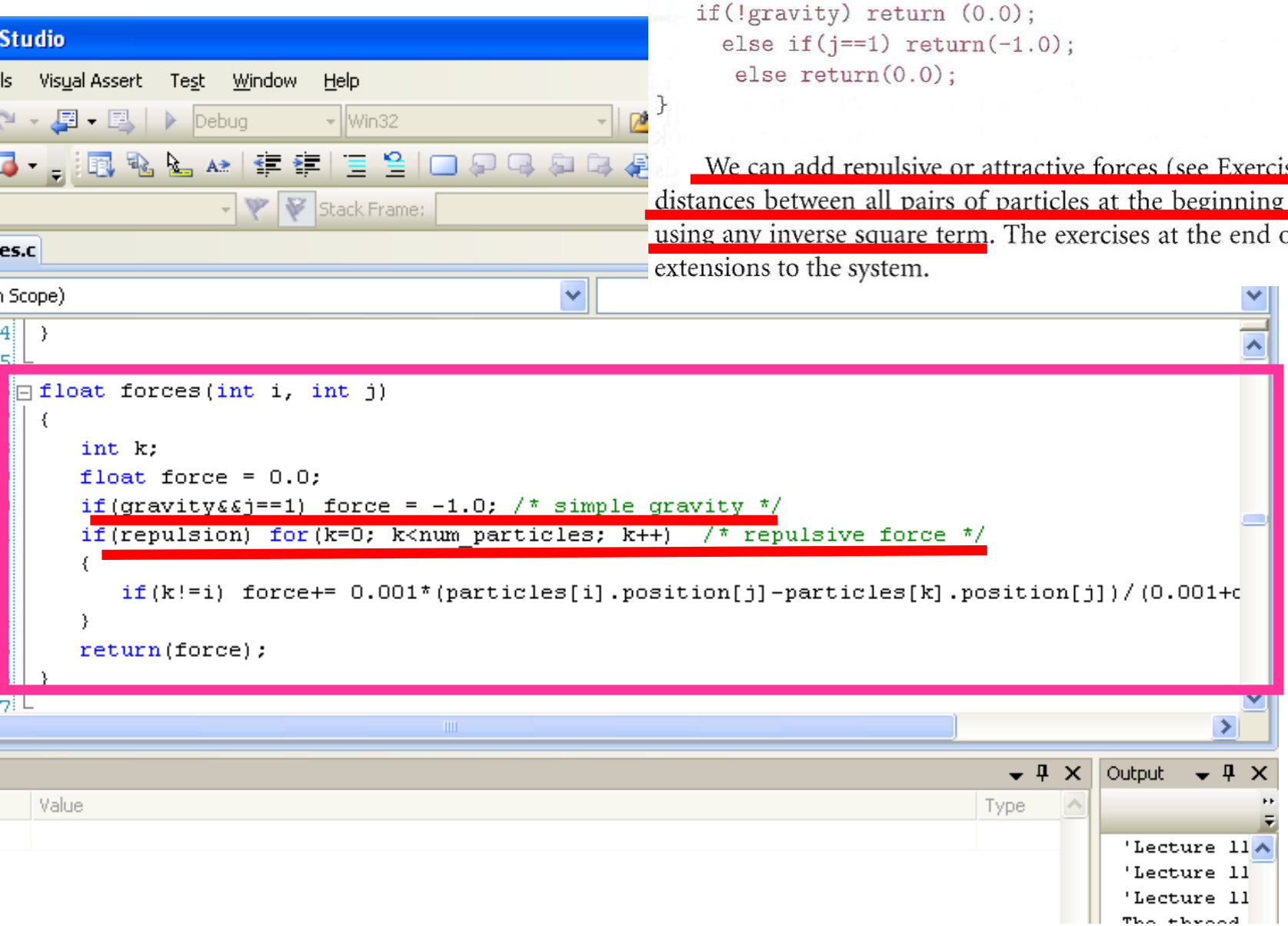
Forces

If the forces are set to zero, the particles will bounce around the box on linear paths continuously. If the coefficient is less than 1.0, eventually the particles will slow to a halt. The easiest force to add is gravity. For example, if all the particles have the same mass, we can add a gravitational term in the y-direction as follows:

```
bool gravity =  TRUE;

float forces(int i, int j)
{
   if(!gravity) return (0.0);
     else if(j==1) return(-1.0);
      else return(0.0);
}
```

We can add repulsive or attractive forces (see Exercise 11.14) by computing the distances between all pairs of particles at the beginning of each iteration and then using any inverse square term. The exercises at the end of the chapter suggest some extensions to the system.



Force Vector

- n Independent Particles
  - Gravity
  - Wind forces
  - O(n) calculation
- n Coupled Particles O(n)
  - Meshes
  - Spring-Mass Systems
- n Coupled Particles $O(n^2)$
  - Attractive and repulsive forces

Simple Forces

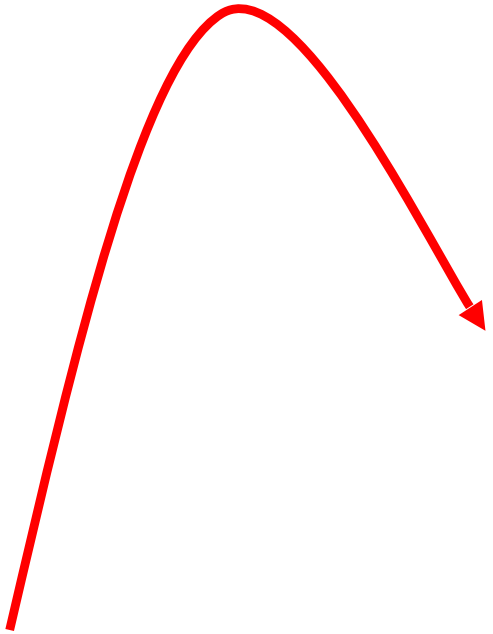
- Consider force on particle i

$$\mathbf{f}_i = \mathbf{f}_i(\mathbf{p}_i, \mathbf{v}_i)$$

- Gravity $\mathbf{f}_i = \mathbf{g}$

$$\mathbf{g}_i = (0, -\mathbf{g}, 0)$$

- Wind forces
- Drag

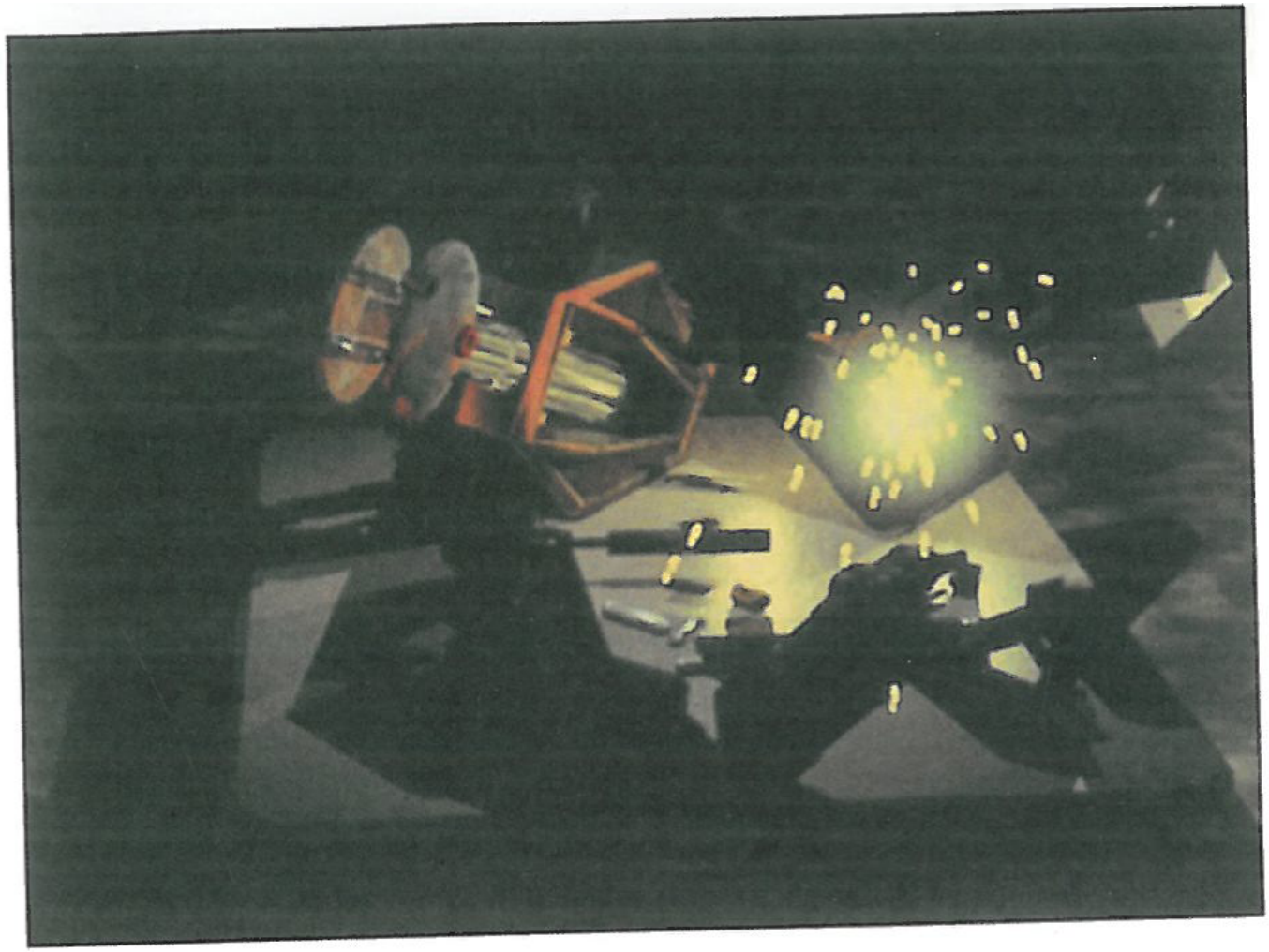


We can simulate phenomena such as fireworks.

Simple Forces

Color Plate 13 Welding scene from video "I Thought, Therefore I Was."

(Courtesy of James Pinkerton, Thomas Keller, Brian Jones, John Bell, University of New Mexico and Sandia National Laboratories.)



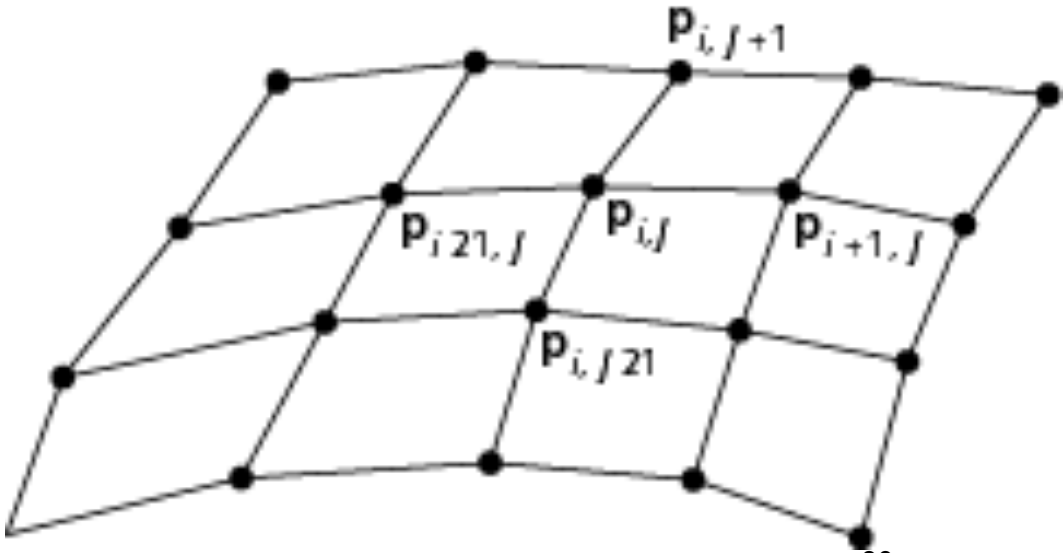
We can simulate phenomena such as fireworks.

Meshes of n particles

- Consider the example of using particles to **create a Surface** whose shape varies over time, such as a Curtain or a Flag blowing in the wind.

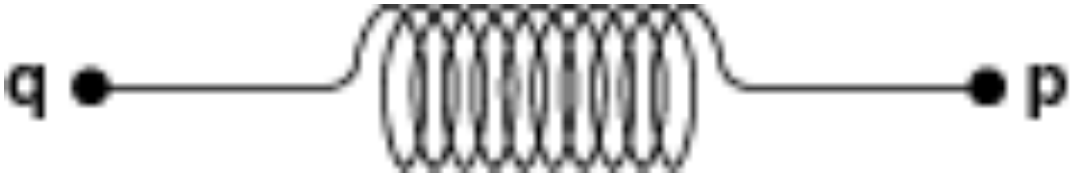
- We can **approximate** this type of force by considering the forces between a particle and the latters closest neighbors.

  O(**n**) force calculation



Spring Forces

- Assume each particle has **unit mass** and is connected to its neighbor(s) by a spring
- Hooke's law: force proportional to distance ($d = \|\mathbf{p} - \mathbf{q}\|$) between the **Points**



$$\mathbf{f} = -k_s(|\mathbf{d}| - s)\frac{\mathbf{d}}{|\mathbf{d}|},$$

Hooke's Law

$$\mathbf{f} = -k_s(|\mathbf{d}| - s)\frac{\mathbf{d}}{|\mathbf{d}|},$$

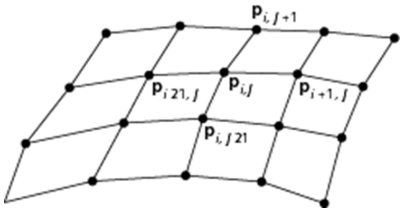
- Let s be the distance when there is **no force**

$$\mathbf{f} = -k_s(|\mathbf{d}| - s)\frac{\mathbf{d}}{|\mathbf{d}|},$$

k_s is the spring constant

$\mathbf{d}/|\mathbf{d}|$ is a **unit vector** pointed from **p** to **q**

- Each **interior point** in **mesh** has **four forces** applied to **it**



Spring Damping

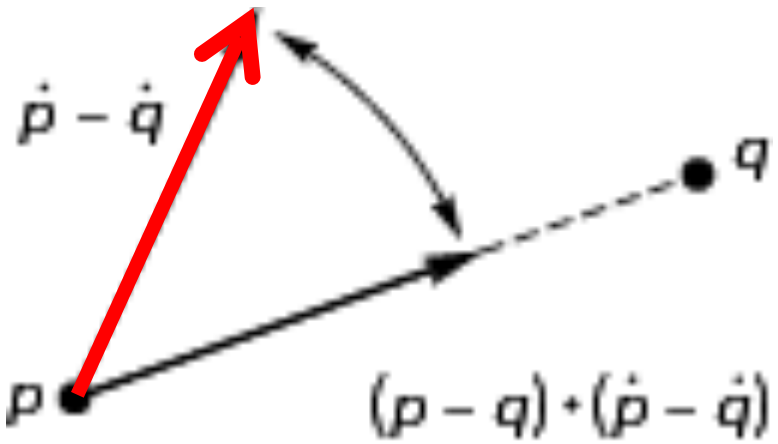
- A pure spring-mass will oscillate forever
- Must add a damping term (drag)

$$\mathbf{f} = -\left(k_s(|\mathbf{d}| - s) + k_d \frac{\dot{\mathbf{d}} \cdot \mathbf{d}}{|\mathbf{d}|}\right) \frac{\mathbf{d}}{|\mathbf{d}|}.$$

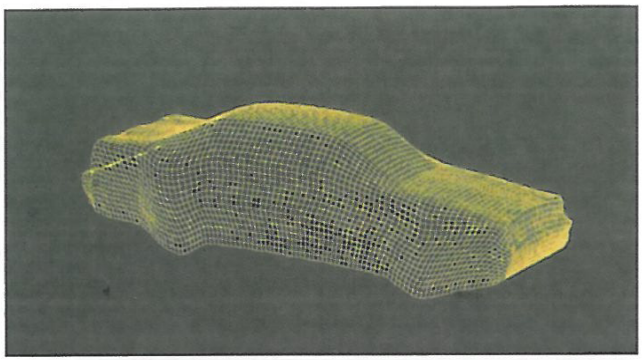
Here, k_d is the damping constant, and

$$\dot{\mathbf{d}} = \dot{\mathbf{p}} - \dot{\mathbf{q}}.$$

- **Must project velocity**



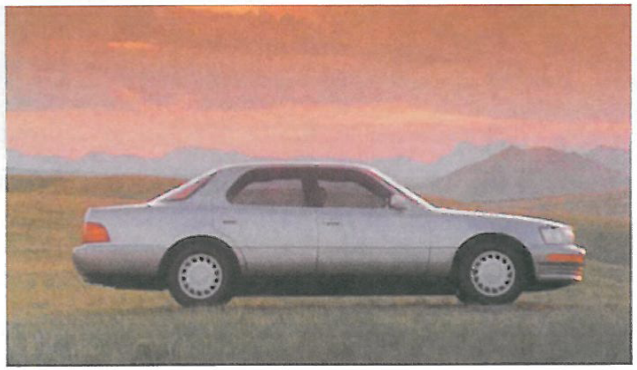
Mesh generated from particles



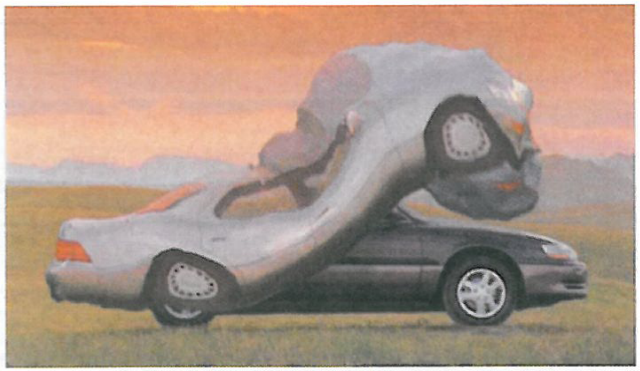
Color Plate 31 Particle system.

(Courtesy of Lexus and Rhythm & Hues.)

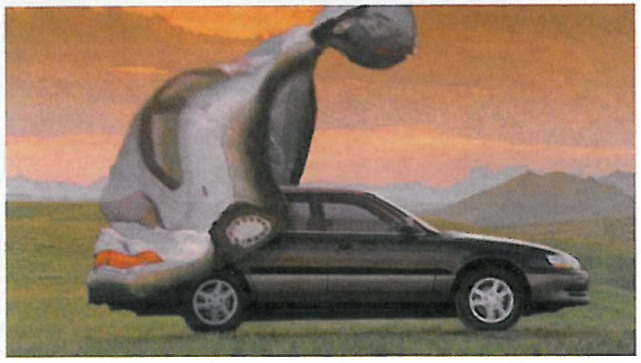
(a) Mesh of particles



(b) Model of Lexus with surface



(c) Wind blowing mesh off Lexus



(c) Mesh blown away from Lexus

Attraction and Repulsion

Whereas spring forces are used to keep a group of particles together, repulsive forces push particles away from one another and attractive forces pull particles toward one another.

We could use repulsive forces to **distribute particles over a surface, or if the particles represent locations of objects, to keep objects from hitting one another**.

We could use attractive forces to build a **model of the solar system or to create applications that model satellites revolving about the earth**.

The equations for attraction and repulsion are essentially the same except for a sign. **Physical models** of particle behavior may include both **attractive** and **repulsive forces**

Attraction and Repulsion

Inverse square law

$$\mathbf{f} = -k_r \frac{\mathbf{d}}{|\mathbf{d}|^3}$$

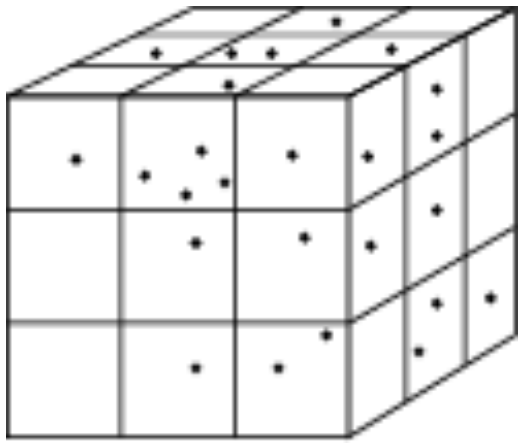
General case requires **O(n^2)** calculation

In most problems, the drop off is such that **not many particles contribute to the forces on any given particle**

Sorting problem: is it **O(n log n)**?

Boxes

- Spatial subdivision technique
- Divide space into **boxes**
- **Particle** can only interact with **particles in its box or the neighboring boxes**
- **Must update which box a particle belongs to after each time step**



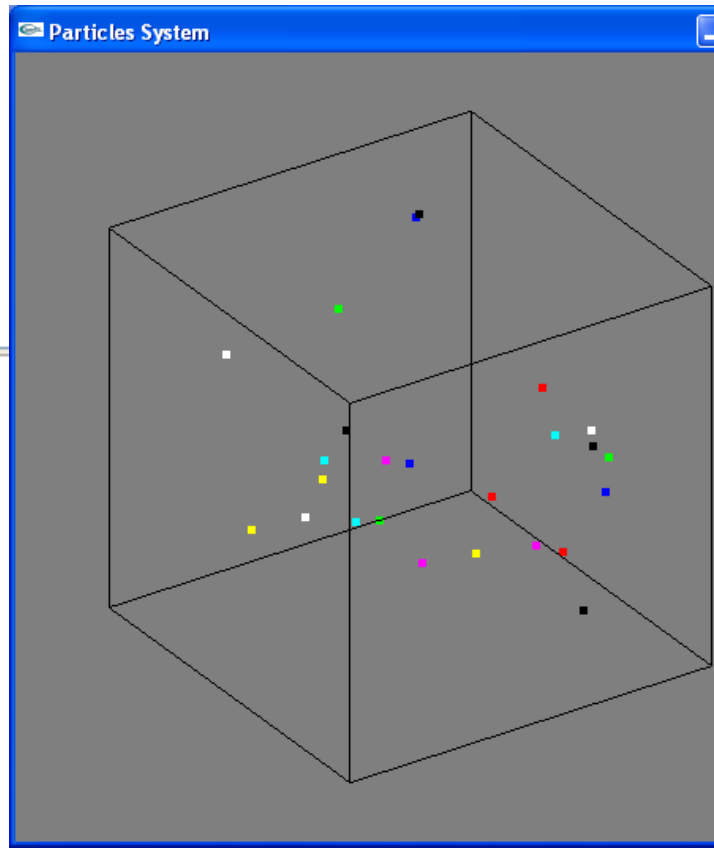
Linked Lists

Each particle maintains a linked **List** of its neighbors

Update data structure at each **time step**

Must amortize **cost of building the data structures** initially

particles.c



(Unknown Scope)

```
22  void myIdle();
23  void myReshape(int, int);
24  void main_menu(int);
25  void collision(int);
26  float forces(int, int);
27
        myinit();

    lobals */

    num_particles; /* number of particles */

    article struct */
35
3   typedef struct particle
3   {
3         int color;
3         float position[3];
4         float velocity[3];
4         float mass;
4
4
4   } particle;
4
4   particle particles[MAX_NUM_PARTICLES]; /* particle system */
4
```

particles.c
Class Participation 1

Particle Field Calculations

- Consider simple gravity
- We don't compute **forces** due to sun, moon, and other large bodies
- Rather we use the gravitational field
- Usually, we can group particles into equivalent **point masses**

The entire particle system can be described by 6n ordinary differential equations of the form $\dot{\mathbf{u}} = \mathbf{g}(\mathbf{u}, t)$, where u is an array of the 6n position and velocity components, g includes the external forces applied to the **particles**.

Solution of ODEs

Particle system has 6n ordinary **differential equations**

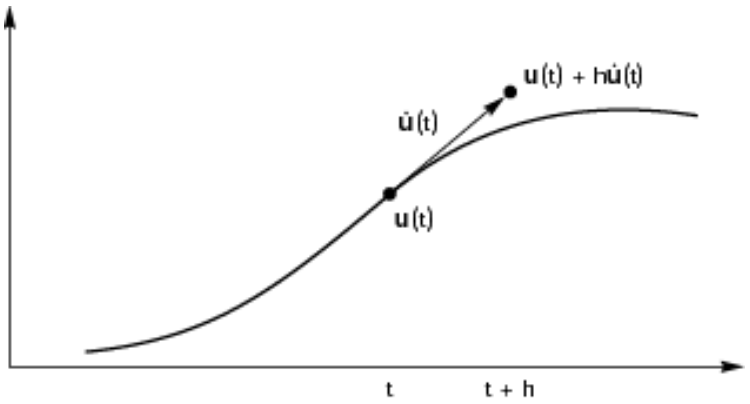
Write set as $d\mathbf{u}/dt = g(\mathbf{u},t)$

Solve by approximations using Taylor's Theorem

If h is small, we can approximate the value of $\mathbf{g}$ over the interval $[t, t + h]$ by the value of $\mathbf{g}$ at t; thus,

$$\mathbf{u}(t + h) \approx \mathbf{u}(t) + h\mathbf{g}(\mathbf{u}(t), t).$$

This expression shows that we can use the value of the derivative at t to get us to an approximate value of **u**(t +h)



Euler's Method

$$\mathbf{u}(t+h) \approx \mathbf{u}(t) + h\mathbf{g}(\mathbf{u}(t), t).$$

Per step error is **O(h²)**

Require one force evaluation per **time step**

Problem is numerical instability

depends on step size

Improved Euler

$$\mathbf{u}(t+h) = \mathbf{u}(t) + \int_t^{t+h} \mathbf{g}(\mathbf{u}, \tau)d\tau.$$

This time, we approximate the integral by an average value over the interval $[t, t+h]$:

$$\int_t^{t+h} \mathbf{g}(\mathbf{u}, \tau)d\tau \approx \frac{h}{2}(\mathbf{g}(\mathbf{u}(t), t) + \mathbf{g}(\mathbf{u}(t+h), t+h)).$$

The problem now is we do not have $\mathbf{g}(\mathbf{u}(t+h), t+h)$; we have only $\mathbf{g}(\mathbf{u}(t), t)$. We can use Euler's method to approximate $\mathbf{g}(\mathbf{u}(t+h), t+h)$; that is, we can use

$$\mathbf{g}(\mathbf{u}(t+h), t+h) \approx \mathbf{g}(\mathbf{u}(t) + h\mathbf{g}(\mathbf{u}(t), t), t+h).$$

Per step error is **O(h^3)**

Also allows for larger step sizes

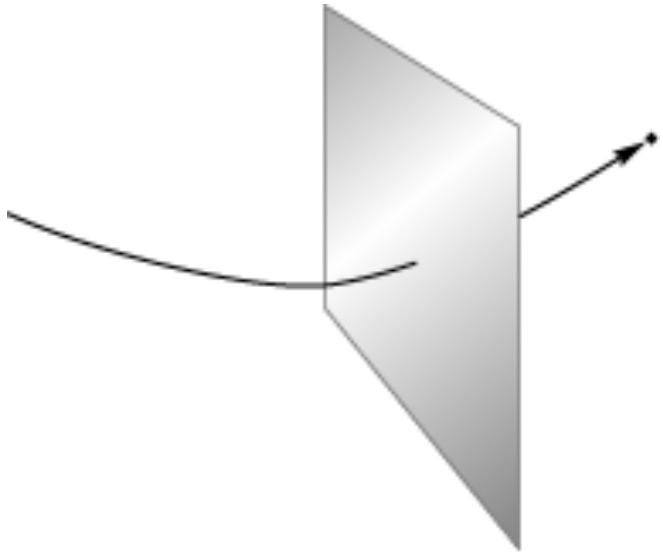
But requires **two function evaluations** per step

Also known as Runge-Kutta method of order 2

Constraints

- Easy in **computer graphics to ignore physical reality (close enough is good enough!)**
- **Surfaces** are virtual

- **Must detect collisions separately if we want exact solution**
- Can approximate with repulsive forces

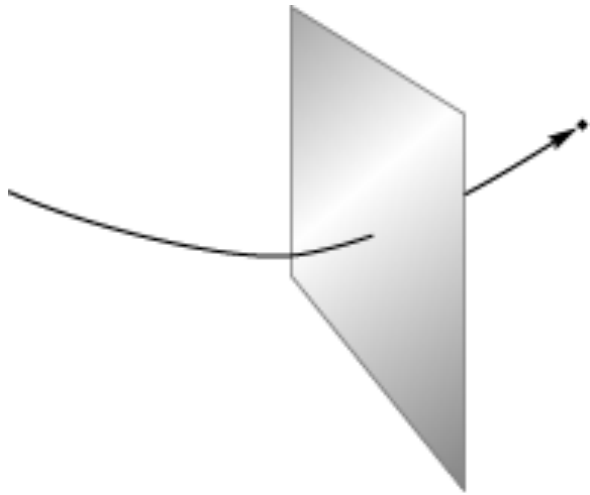


Constraints

Must detect collisions separately if we want exact solution

Suppose that one of the particles has penetrated a polygon, as shown in the figure below.

We can detect this collision by inserting the position of the particle into the **equation of the Plane of the polygon**. If the **time step** of our differential equation solver is small, we can assume that the **velocity** between **time steps** is constant, and we can **use linear interpolation to find the time at which the particle actually hit the Polygon**



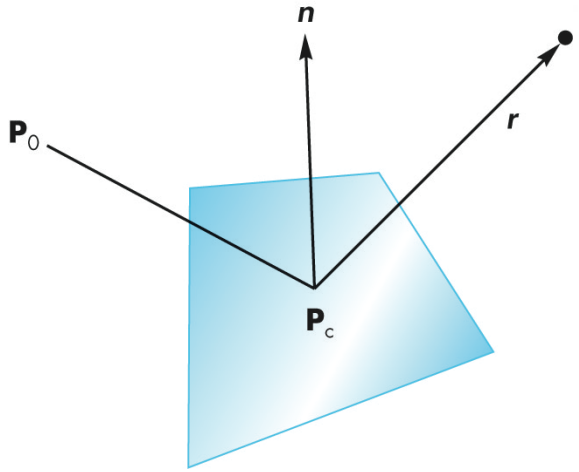
Collisions

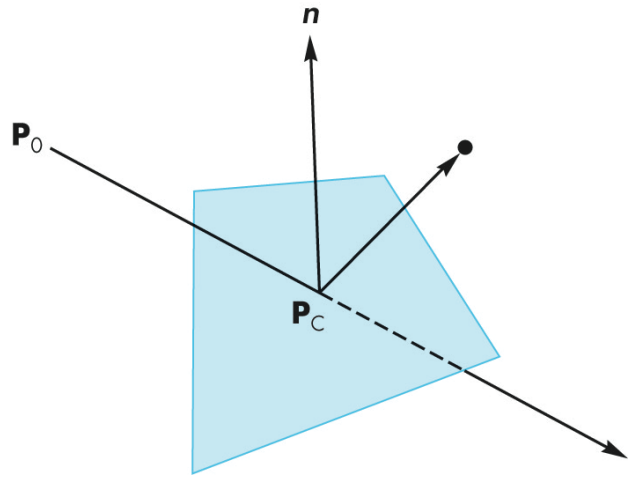
Once we detect a collision, **we can calculate new path**

Reflect vertical component

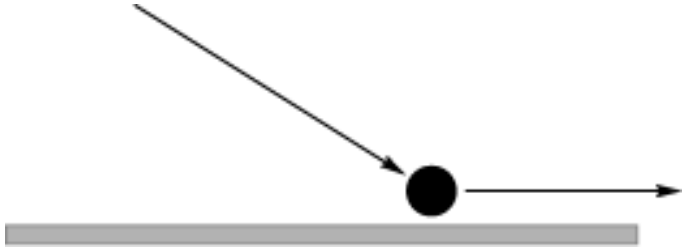
However, its direction after the collision is in the direction of a perfect reflection. Thus, given the **normal at the point of collision Pc** and the previous position of the **particle P0**, we can compute the **direction of a perfect reflection**, as we did in Chapter 6, using the **vector from the particle** to the surface and the **normal** at the surface.

$$\mathbf{r} = 2(\mathbf{p}_0 - \mathbf{p}_c) \cdot \mathbf{n}\ \mathbf{n} - (\mathbf{P}_0 - \mathbf{P}_c).$$





Contact Forces



Suppose that we have a particle that is subject to a force pushing it along a surface, as shown in the figure. The particle cannot penetrate the surface, and it cannot bounce from the surface because of the force being applied to it.

We can argue that the particle is subject to the tangential component of the **applied force** that is, **the part of** the applied force along the surface. We can also apply frictional terms in this direction.

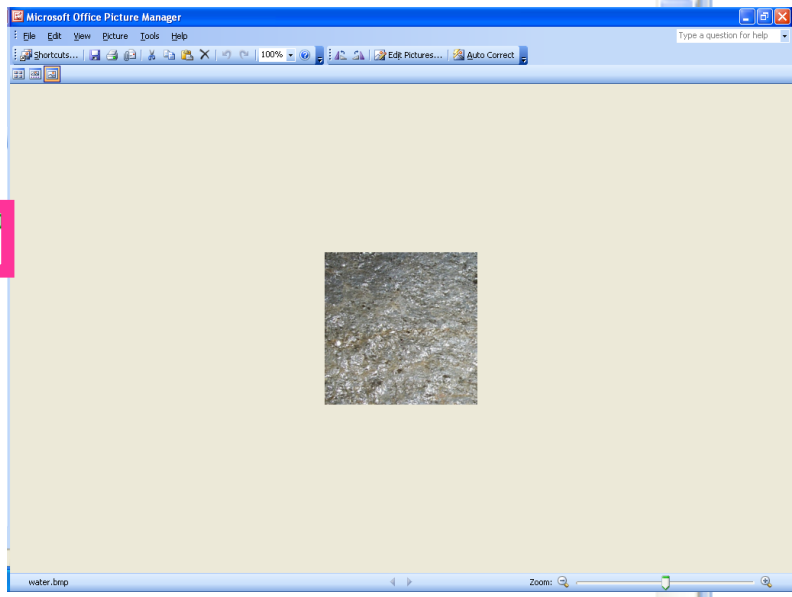
Particle Systems Example
Fountain Simulation

fountain.cpp*

(Unknown Scope)

```
87  class CDrop
88  {
89  private:
90      GLfloat time;           // How many steps the drop was "outside", whe
91                              // falls into the water, time is set back to 0
92      SVertex ConstantSpeed;
93      GLfloat AccFactor;
94  public:
95      void SetConstantSpeed (SVertex NewSpeed);
96      void SetAccFactor(GLfloat NewAccFactor);
97      void SetTime(GLfloat NewTime);
98      void GetNewPosition(SVertex * PositionVertex);  //i
99  };
100
101 void CDrop::SetConstantSpeed(SVertex NewSpeed)
102 {
103     ConstantSpeed = NewSpeed;
104 }
105
106 void CDrop::SetAccFactor (GLfloat NewAccFactor)
107 {
108     AccFactor = NewAccFactor;
109 }
110
111 void CDrop::SetTime(GLfloat N
112 {
```

water.bmp



fountain.cpp

fountain.cpp
Class Participation 2

Newtonian Particles

We conclude our development of partial systems by building a simple particle system that can be expanded to more complex behaviors. Our particles are all Newtonian so their state is described by their positions and velocities. In addition, each particle can have its own color and mass. We start with the following structure:

```
typedef struct particle
{
     int color;
     float position[3];
     float velocity[3];
     float mass;
} particle;
```

A particle system is an array of particles:

```
particle particles[MAX_NUM_PARTICLES];
```

fountain.cpp

(Unknown Scope)

```
class CDrop
{
private:
     GLfloat time;           // How many steps the drop was "outside", when it
                             // falls into the water, time is set back to 0
     SVertex ConstantSpeed;
     GLfloat AccFactor;
public:
     void SetConstantSpeed (SVertex NewSpeed);
     void SetAccFactor(GLfloat NewAccFactor);
     void SetTime(GLfloat NewTime);
     void GetNewPosition(SVertex * PositionVertex);  //increments time, gets the
};

void CDrop::SetConstantSpeed(SVertex NewSpeed)
{
     ConstantSpeed = NewSpeed;
}

void CDrop::SetAccFactor (GLfloat NewAccFactor)
{
     AccFactor = NewAccFactor;
}

void CDrop::SetTime(GLfloat NewTime)
{
     time = NewTime;
}
```

Displaying Particles

Given the position of a particle, we can display it using any set of primitives, either geometric or raster, that we like. A simple starting point would be to display each particle as a point. Here is a display callback that loops through the array of particles:

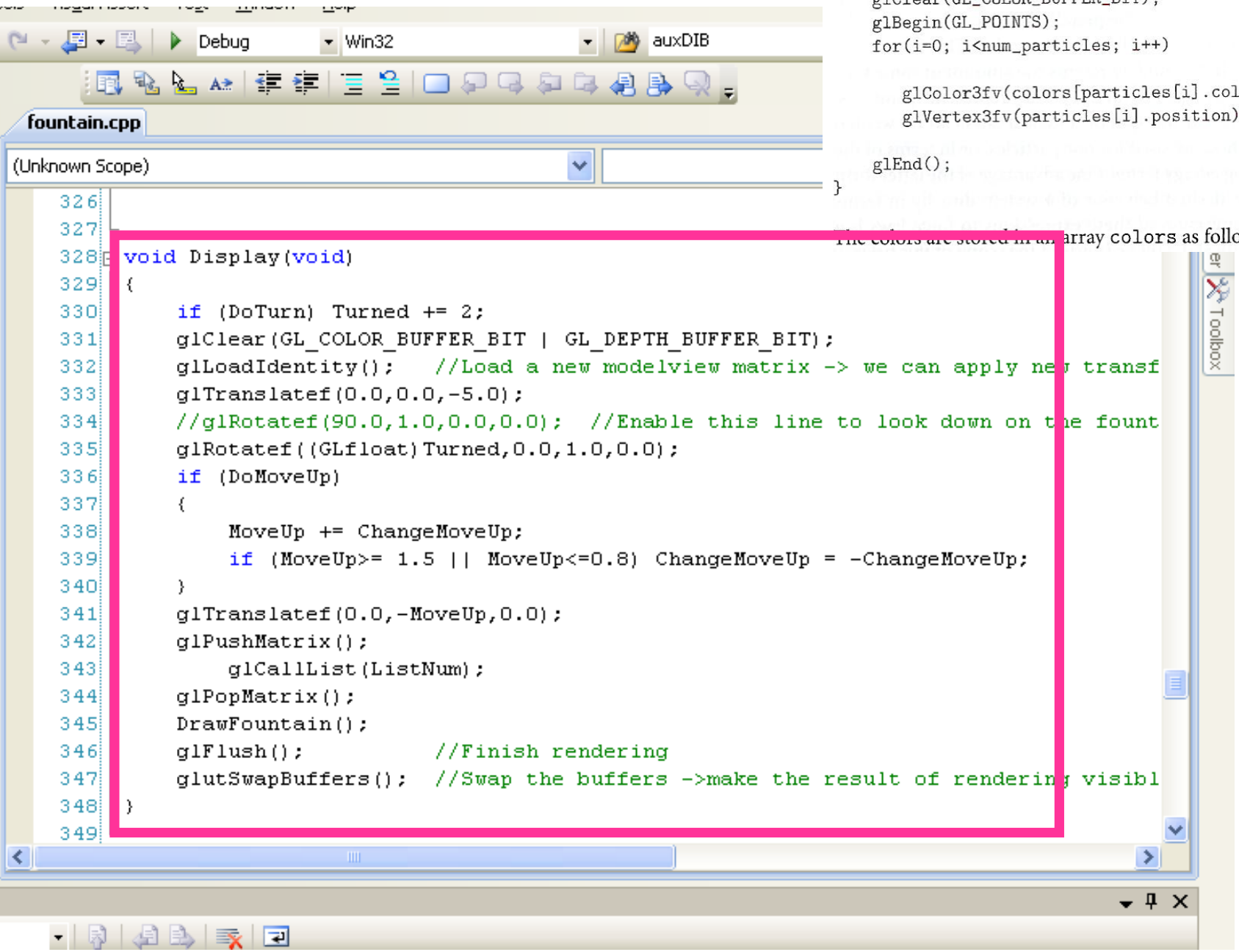
```
void myDisplay()
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
    for(i=0; i<num_particles; i++)

       glColor3fv(colors[particles[i].color]);
       glVertex3fv(particles[i].position);

    glEnd();
}
```

The colors are stored in an array colors as follows:



Updating Particle Positions

```
float last_time, present_time;
float num_particles;

void myIdle()
{
    int i, j;
    float dt;
    present_time = glutGet(GLUT_ELAPSED_TIME); /* in milliseconds */
    dt = 0.001*(present_time -  last_time); /* in seconds */
    for(i=0; i<num_particles; i++)
    {
       for(j=0; j<3; j++)
       {
           particles[i].position[j]+=dt*particles[i].velocity[j];
           particles[i].velocity[j]+=dt*forces(i,j)/particles[i].mass;
       }
        collision(i);
    }
    last_time = present_time;
    glutPostRedisplay();
}
```

tudio

ebug Tools Visual Assert Test Window Help

Debug Win32 auxDIB

fountain.cpp

(Unknown Scope)

```
115
116 void CDrop::GetNewPosition(SVertex * PositionVertex)
117 {
118     SVertex Position;
119     time += 1.0;
120     Position.x = ConstantSpeed.x * time;
121     Position.y = ConstantSpeed.y * time - AccFactor * time * time;
122     Position.z = ConstantSpeed.z * time;
123     PositionVertex->x = Position.x;
124     PositionVertex->y = Position.y + WaterHeight;
125     PositionVertex->z = Position.z;
126     if (Position.y < 0.0)
127     {
128         /*the drop has fallen into the water. The problem is now, that we cannot
129         set time to 0.0, because if there are more "DropsPerRay" than "TimeNeeded" (See
130         several drops would be seen as one. Check it out.
131         */
132         time = time - int(time);
133         if (time > 0.0) time -= 1.0;
134     }
135
136 }
137
```

Initialization

```
float num_particles = INITIAL_NUM_PARTICLES;
float speed = INITIAL_SPEED;

void myinit()
{
     int  i, j;
     for(i=0; i<num_particles; i++)
     {
          particles[i].mass = 1.0; /* default particle mass */
          particles[i].color = i%8; /* 8 colors in color array */
          for(j=0; j<3; j++)
          {
               particles[i].position[j] = 2.0*((float) random()/
                  RAND_MAX)-1.0;
               particles[i].velocity[j] = speed*2.0*((float) random()/
                  RAND_MAX)-1.0;
          }
     }
}
```

```
void InitFountain(void)
{
    //This function needn't be and isn't speed optim
    FountainDrops = new CDrop [ DropsComplete ];
    FountainVertices = new SVertex [ DropsComplete ]
    SVertex NewSpeed;
    GLfloat DropAccFactor; //different from AccFacto
    GLfloat TimeNeeded;
    GLfloat StepAngle; //Angle, which the ray gets out of the fountain with
    GLfloat RayAngle;   //Angle you see when you look down on the fountain
    GLint i,j,k;
    for (k = 0; k <Steps; k++)
    {
        for (j = 0; j < RaysPerStep; j++)
        {
            for (i = 0; i < DropsPerRay; i++)
            {
                DropAccFactor = AccFactor + GetRandomFloat(0.0005);
                StepAngle = AngleOfDeepestStep + (90.0-AngleOfDeepestStep)
                        * GLfloat(k) / (Steps-1) + GetRandomFloat(0.2+0.8*(Steps-k-1)/(Steps
                //This is the speed caused by the step:
                NewSpeed.x = cos ( StepAngle * PI / 180.0) * (0.2+0.04*k);
                NewSpeed.y = sin ( StepAngle * PI / 180.0) * (0.2+0.04*k);
                //This is the speed caused by the ray:
                RayAngle = (GLfloat)j / (GLfloat)RaysPerStep * 360.0;
                //for the next computations "NewSpeed.x" is the radius. Care! Dont swap the
                //lines, because the second one changes NewSpeed.x!
                NewSpeed.z = NewSpeed.x * sin ( RayAngle * PI /180.0);
                NewSpeed.x = NewSpeed.x * cos ( RayAngle * PI /180.0);
                //Calculate how many steps are required, that a drop comes out and falls dow
                TimeNeeded = NewSpeed.y/ DropAccFactor;
                FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetConstantSpeed (
```

fountain.cpp

Collisions

We use the collision function to keep the particles inside the initial axis-aligned box. Our strategy is to increment the position of each particle and then check to see if the particle has crossed one of the sides of the box. If it has crossed a side then we can treat the bounce as a reflection. Thus, we need only to change the sign of the velocity in the normal direction and place the particle on the other side of the box. If the coefficient of restitution is less than 1.0, the particles will slow down when they hit a side of the box:

```
float coef; /* coefficient of restitution */

void collision(int n)
{
   int i;
   for (i=0; i<3; i++)
   {
      if(particles[n].position[i]>=1.0)
      {
         particles[n].velocity[i] = -coef*particles[n].velocity[i];
         particles[n].position[i] =
                                 1.0-coef*(particles[n].position[i]-1.0);
      }
      if(particles[n].position[i]<=-1.0)
      {
        particles[n].velocity[i] = -coef*particles[n].velocity[i];

        particles[n].position[i] = -1.0-
                                   coef*(particles[n].position[i]+1.0);
      }
   }
}
```

fountain.cpp

Forces

If the forces are set to zero, the particles will bounce around the box on linear paths continuously. If the coefficient is less than 1.0, eventually the particles will slow to a halt. The easiest force to add is gravity. For example, if all the particles have the same mass, we can add a gravitational term in the y-direction as follows:

```
bool gravity =  TRUE;

float forces(int i, int j)
{
   if(!gravity) return (0.0);
     else if(j==1) return(-1.0);
      else return(0.0);
}
```

We can add repulsive or attractive forces (see Exercise 11.14) by computing the distances between all pairs of particles at the beginning of each iteration and then using any inverse square term. The exercises at the end of the chapter suggest some extensions to the system.

```
SVertex * Vertices = new SVertex[NumOfVerticesSt
ListNum = glGenLists(1);
for (i = 0; i<NumOfVerticesStone; i++)
{
    Vertices[i].x = cos(2.0 * PI / NumOfVertices
    Vertices[i].y = StoneHeight;  //Top
    Vertices[i].z = sin(2.0 * PI / NumOfVerticesStone * i) * OuterRadius;
}
for (i = 0; i<NumOfVerticesStone; i++)
{
    Vertices[i + NumOfVerticesStone*1].x = cos(2.0 * PI / NumOfVerticesStone * i) * Inne
    Vertices[i + NumOfVerticesStone*1].y = StoneHeight;  //Top
    Vertices[i + NumOfVerticesStone*1].z = sin(2.0 * PI / NumOfVerticesStone * i) * Inne
}
for (i = 0; i<NumOfVerticesStone; i++)
{
    Vertices[i + NumOfVerticesStone*2].x = cos(2.0 * PI / NumOfVerticesStone * i) * Oute
    Vertices[i + NumOfVerticesStone*2].y = 0.0;  //Bottom
    Vertices[i + NumOfVerticesStone*2].z = sin(2.0 * PI / NumOfVerticesStone * i) * Oute
}
for (i = 0; i<NumOfVerticesStone; i++)
{
    Vertices[i + NumOfVerticesStone*3].x = cos(2.0 * PI / NumOfVerticesStone * i) * Inne
    Vertices[i + NumOfVerticesStone*3].y = 0.0;  //Bottom
    Vertices[i + NumOfVerticesStone*3].z = sin(2.0 * PI / NumOfVerticesStone * i) * Inne
```

SUMMARY and NOTES

Procedural methods have advantages in that we can control how many primitives we produce and at which point in the process these primitives are generated. Equally important is that procedural graphics provides an object- oriented approach to building models an approach that should be of increasing importance in the future.

Combining physics with computer graphics provides a set of techniques that has the promise of generating physically-correct animations and of providing new modeling techniques. Particle systems are but one example of physically-based modeling, but they represent a technique that has wide applicability. **One of the most interesting and informative exercises that you can undertake at this point is to build a particle system.**

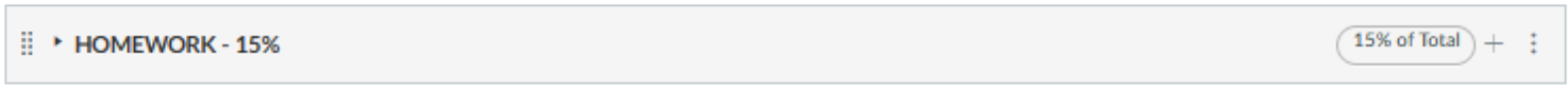
Particle methods are used routinely in commercial animations, both for simulation of physical phenomena, such as fire, clouds, and moving water, and in determining the positions of animated characters. **Particle systems have also become a standard approach to simulating physical phenomena, often replacing complex partial differential equation models, and are used even if a graphical result is not needed.**

Fractals provide another method for generating objects with simple algorithms and programs that produce images that appear to have great complexity. Procedural noise has been at the heart of almost all procedural-modeling methods and its true power is often best demonstrated when it is combined with one or more of the other methods that we have presented.

As we look ahead, we see a further convergence of graphics methods with methods from physics, mathematics, and other sciences. Historically, given the available computing power, we were content to accept visuals that looked okay but were not especially close to the correct physics in applications such as simulation and interactive games. Even in applications in which we might spend days rendering a single frame, the true physics was still too complex to simulate well. **However, with the continued advances in available computing power and the lowered cost of accessing such power, we expect to see more and more physically correct modeling in all applications of computer graphics.**

NEXT.

11.13.2023 (M 5:30 to 7) (24)	Homework 8	**Lecture 12** **(Curves and Surfaces)**
11.15.2023 (W 5:30 to 7) (25)		**Lecture 13** **(Advanced Rendering)**
11.20.2023 (M 5:30 to 7) (26)		**PROJECT 4**
11.27.2023 (M 5:30 to 7) (27)		**EXAM 4 REVIEW**
11.29.2023 (M 5:30 to 7) (28)		**EXAM 4**
12.11.2023 (M 5:30 to 7)		**FINAL EXAM**



At 6:45 PM.

End Class 23

VH, Download Attendance Report
Rename it:
11.08.2023 Attendance Report FINAL

VH, upload Lecture 11 to CANVAS.