01418585
Rendering and Shading Techniques

Lecture 01

Administrivia

About

- This course is a survey course on **rendering algorithms**.
- You are expected to **implement** some of them.
- The aims of the course are:
  - To equip you with **knowledge for future research**.
  - To develop your **programming skills**.

Instructor

- Pramook Khungurn
    - Email: pramook@gmail.com or fscipmk@ku.ac.th
    - Cellphone: 08-5453-5857
    - Office: Numberless room in front of the Department's office
    - Office Hour: Wednesday & Friday 1PM - 4PM or by appointment

Grading

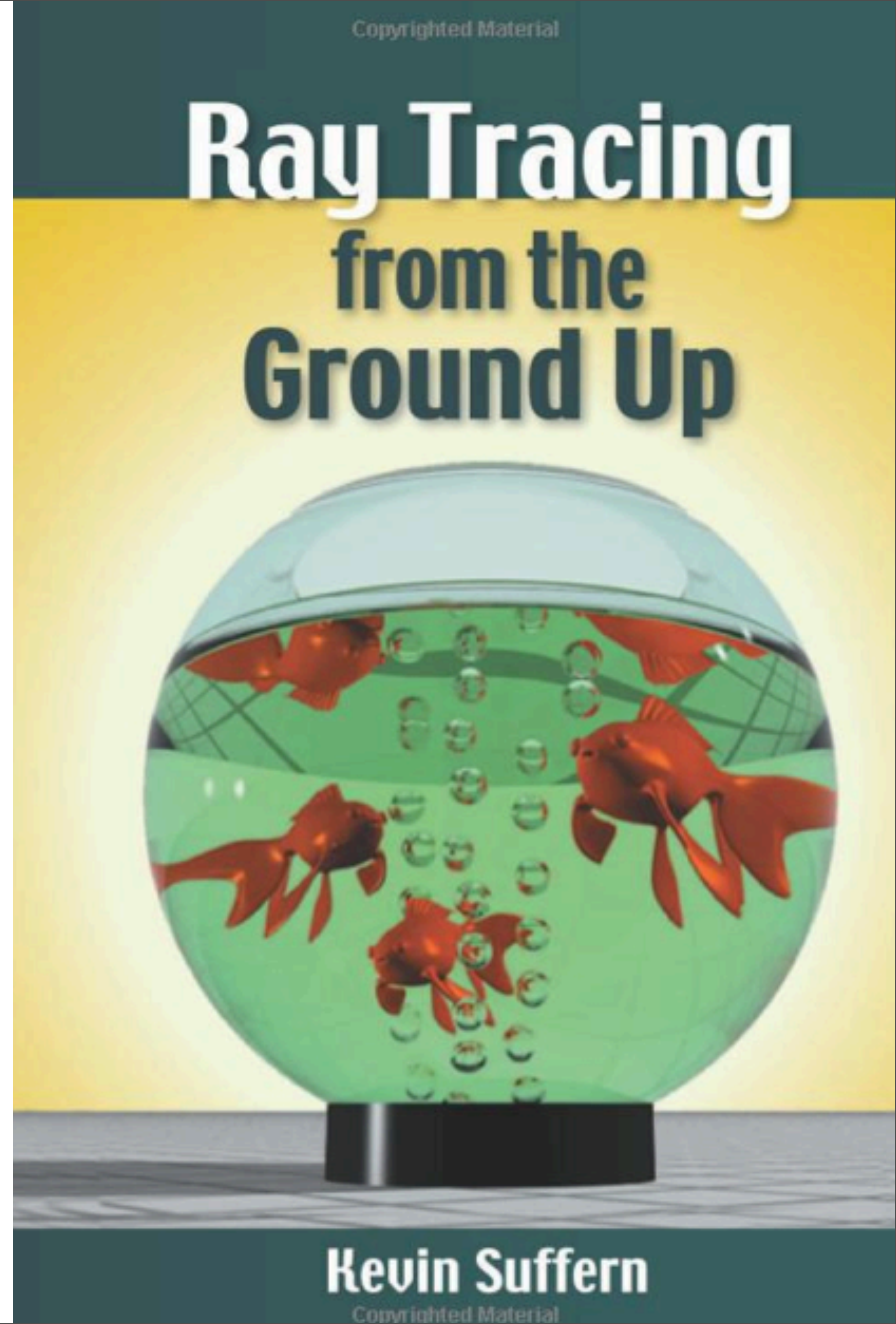
- Homework: 60%
- Final Project: 40%
- No exams.

Requirement

- You should be fluent in C++ (not C).
- You should know:
  - Linear algebra
  - Calculus
  - Probability theory (esp. random variables)

Books

- Kevin Suffern.
  Ray Tracing from the Ground Up.
  A K Peters, 2009.
  - Required
  - Since there will be few students, please order a copy yourself from Amazon or local bookstores.



Books

- Not required
  - Matt Pharr and Greg Humphreys.
    Physically Based Rendering: From Theory to Implementation.
    Elsevier, 2004.
  - Philip Dutre, Kavita Bala, and Philippe Bekaert.
    Advanced Global Illumination.
    A K Peters, 2006.
  - Henrik Wann Jensen.
    Realistic Image Synthesis Using Photon Mapping.
    A K Peters, 2009

Web Page

- http://theory.cpe.ku.ac.th/~pramook/418585/
- Please check it frequently for:
  - Slides
  - Homeworks
- I don’t distribute printouts of slides in class.

Academic Honesty Policy

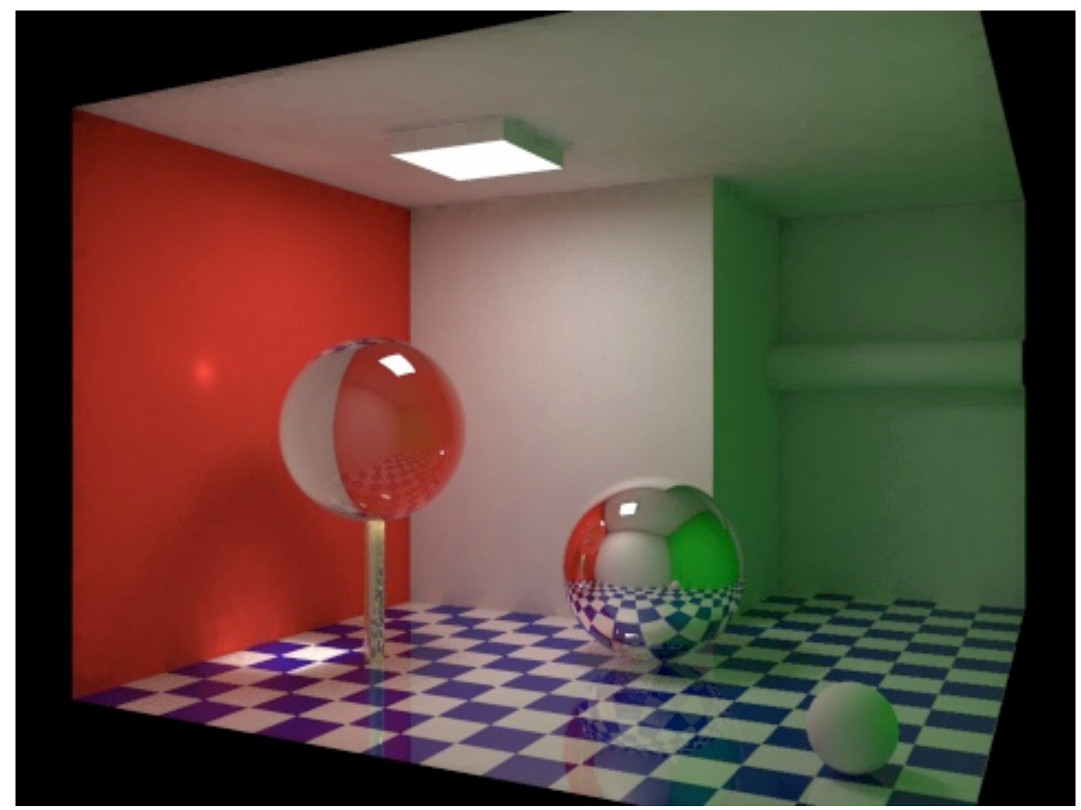
- You shall do all of your homework by yourself.
    - Type your programs yourself.
- Do not plagiarize.
    - Do not copy from your friend or internet sources.
    - If you do, you will earn no credits for the assignment.
- However, feel free to collaborate and consult the internet for ideas.
    - Please also indicate where you get your ideas from in your hand-ins.

Rendering

Rendering

- The process of generating **images** from **models**.

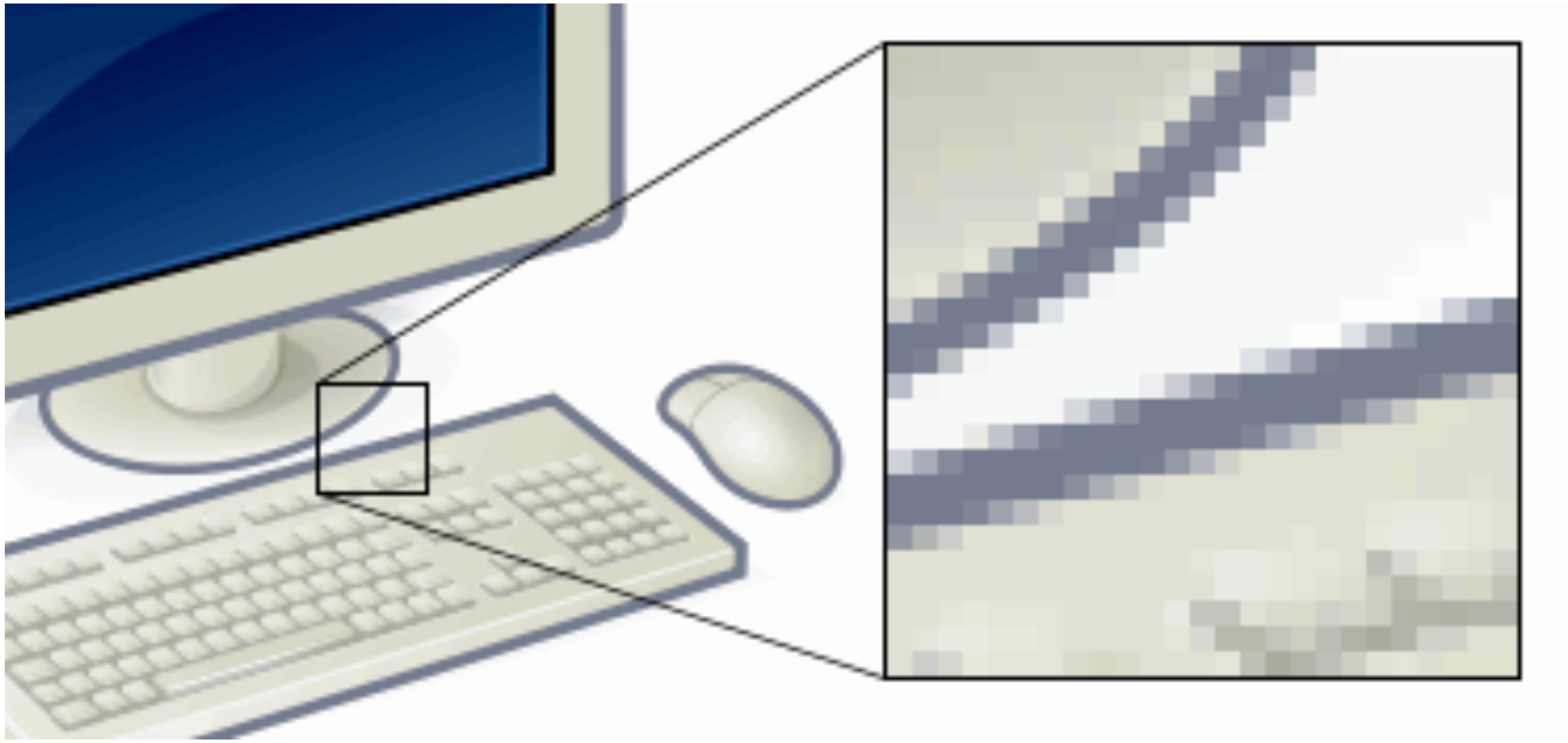
40.3765	246.3446	-13.3601
41.7488	226.0027	-5.0658
48.3294	235.3752	-7.3497
37.2949	230.1558	-9.6773
46.8526	239.2049	-10.7724
35.0925	232.2118	-10.9210
49.2234	231.9015	-5.4622
39.5274	227.7154	-6.8570
36.7923	240.2518	-18.0725
40.9546	241.5318	-16.3400
53.2942	227.1024	-17.4600
51.4157	231.8651	-20.9840
45.7685	234.6469	-25.0268
32.3952	239.7475	-5.4070
36.2495	235.5937	-5.3574
31.0568	236.1462	-9.5742
34.1015	253.4861	-8.2545
31.5805	251.6262	-9.3695
33.9048	256.8511	-4.1244



http://en.wikipedia.org/wiki/Global_illumination

Images

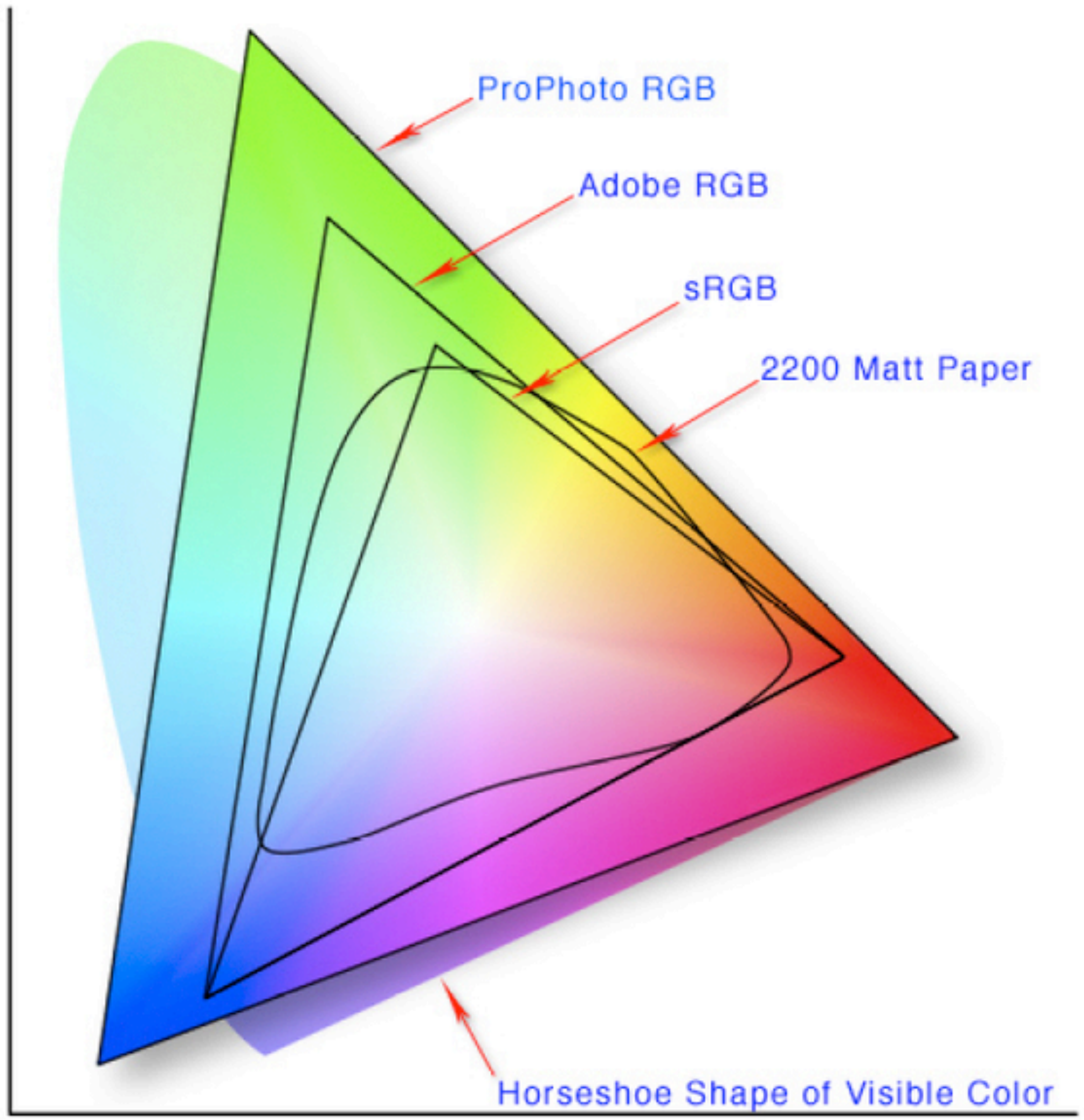
- Rectangular array of squares colors.
- Each square is call a **pixel** (picture element).



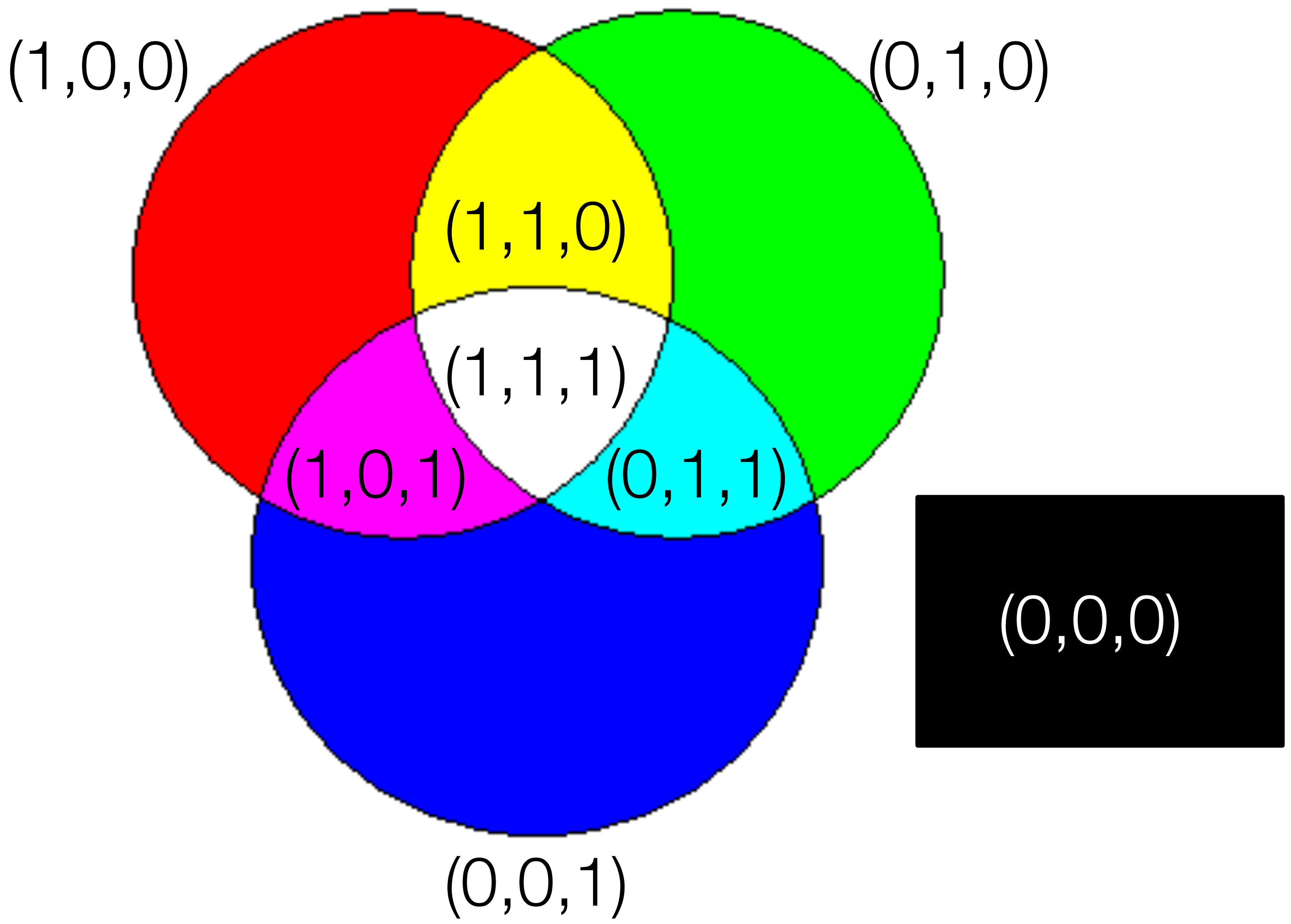
http://en.wikipedia.org/wiki/Pixels

Colors

- A vector (R,G,B)
  - R, G, B are real numbers ranging from 0 to 1
  - They are intensities of the red, green, and blue channel, respectively.

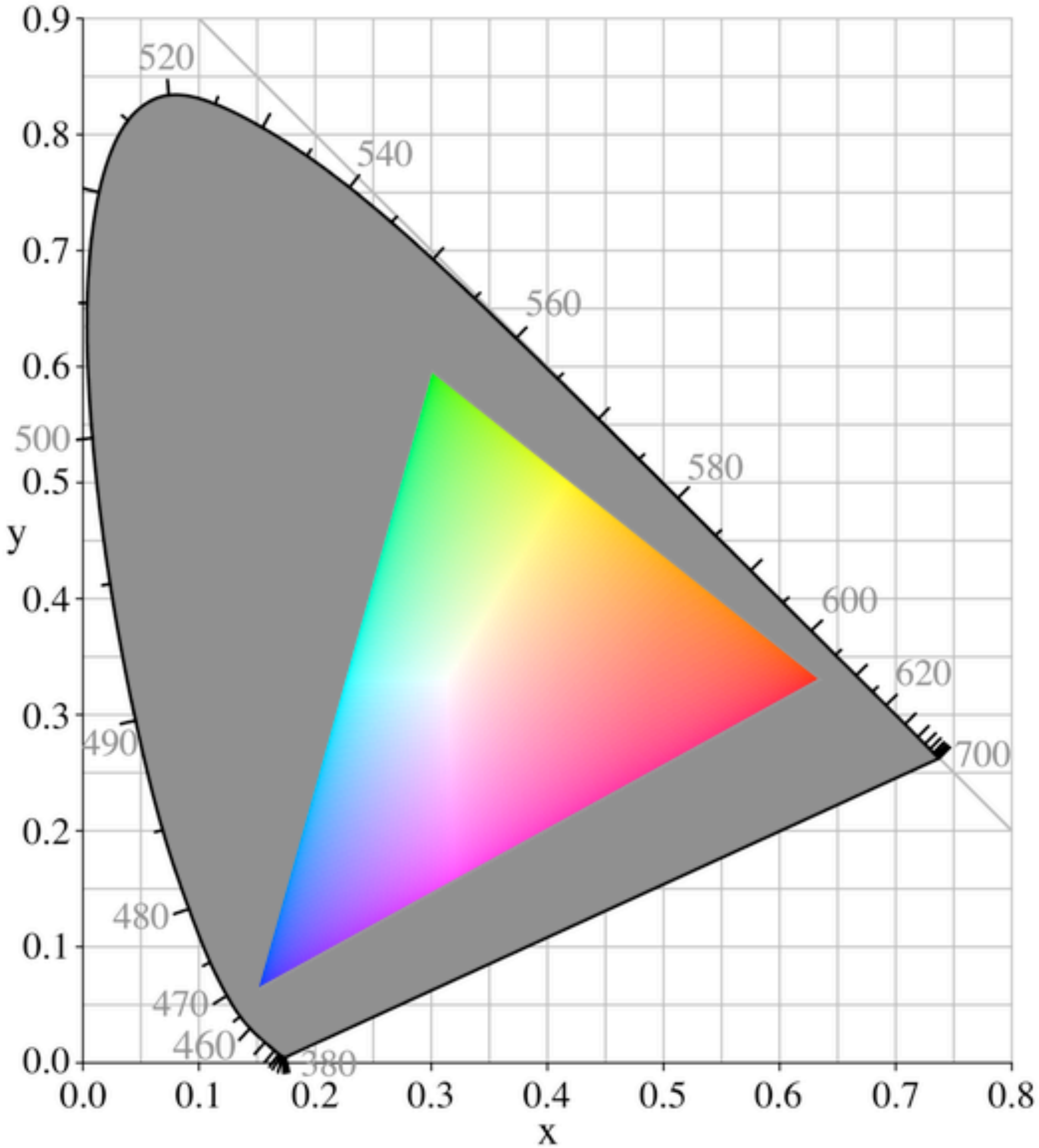


Important Colors and Their RGB Representation



Gamut

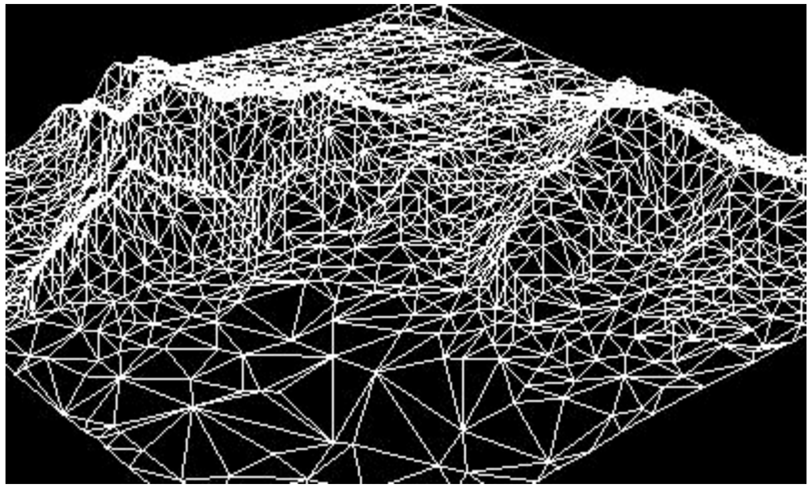
- The range of color a display device can display accurately.
- Namely, all the colors that gets displayed when you set R, G, and B to various values in range [0,1]

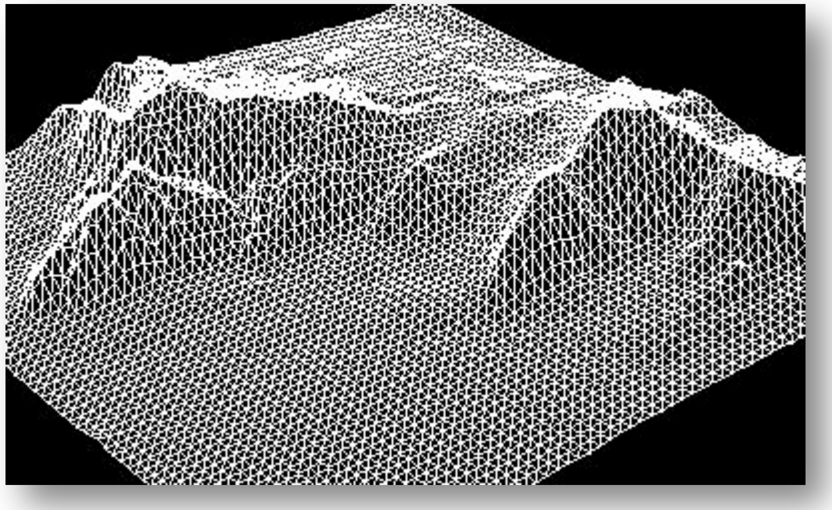


http://upload.wikimedia.org/wikipedia/commons/d/d3/CIExy1931_srgb_gamut.png

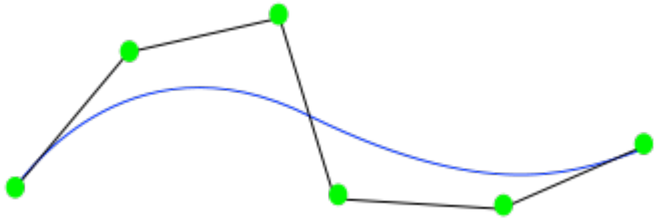
Models

- Mathematical representation of
  - Shapes
  - Optical characteristic of surfaces.





http://amber.rc.arizona.edu/dx/vtkDecimateDX.html



http://en.wikipedia.org/wiki/Nurbs

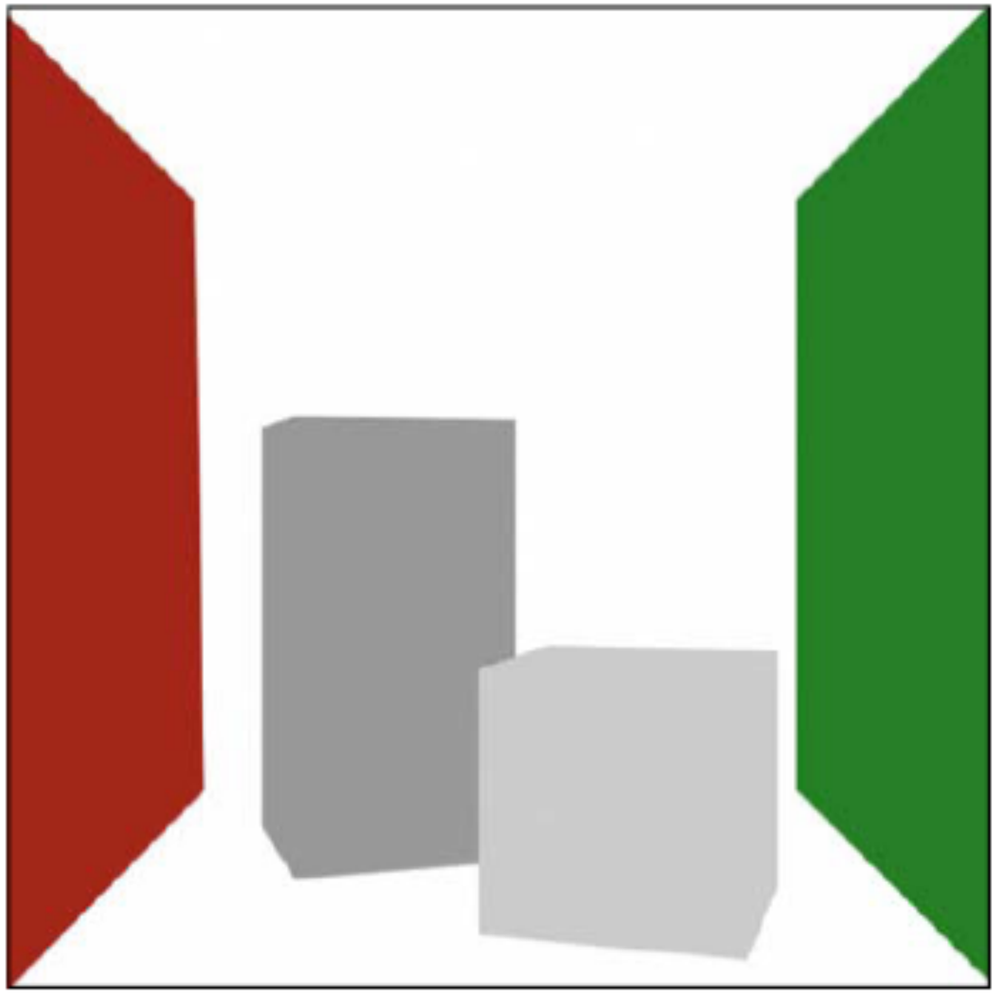
Photorealistic Rendering

http://en.wikipedia.org/wiki/Rendering

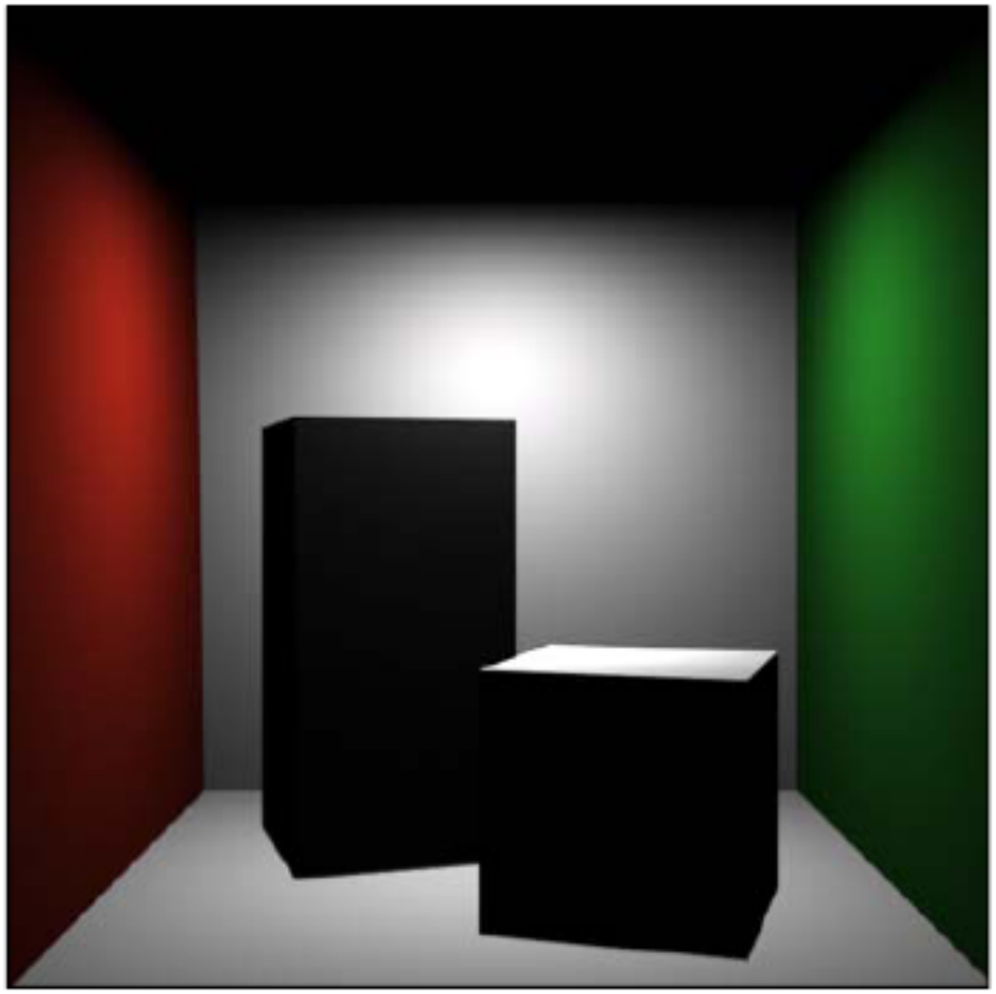
Non-Photorealistic Rendering

The Legend of Zelda: The Wind Waker
http://en.wikipedia.org/wiki/Toon_shading

Lighting: Diffuse Reflection

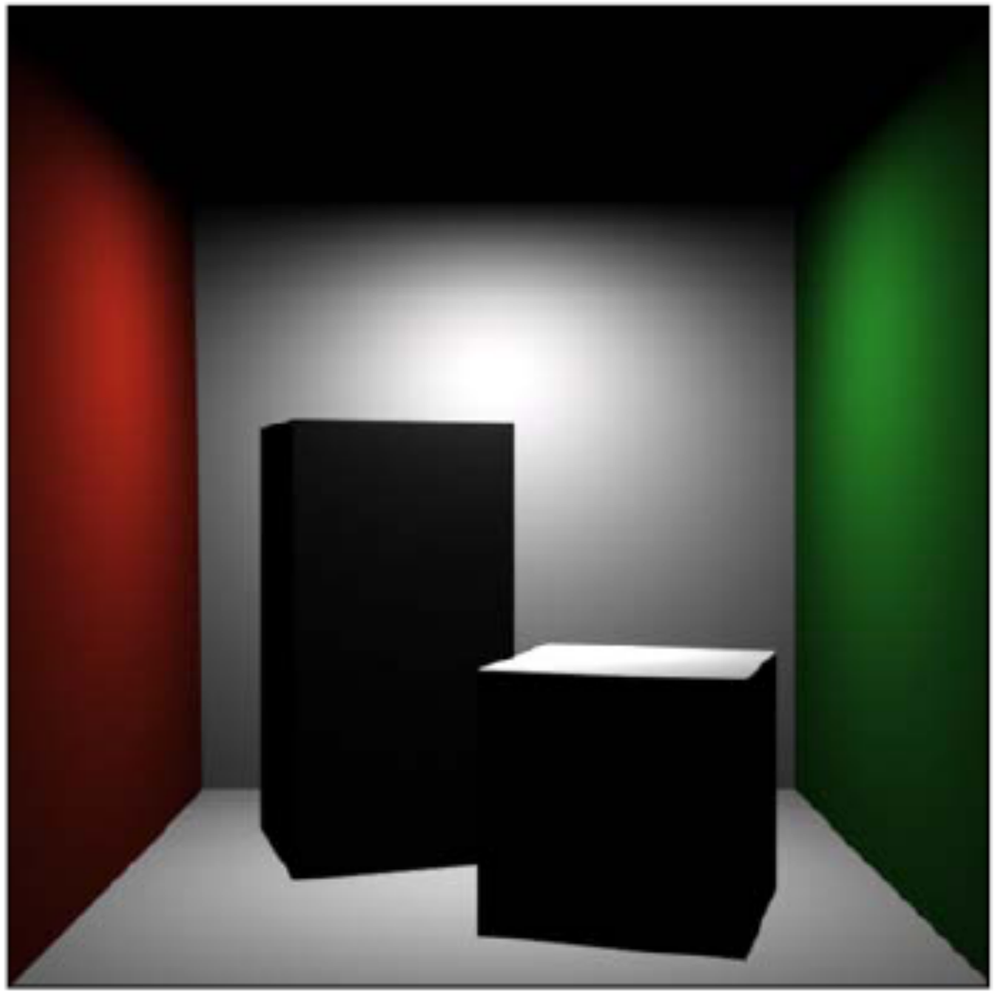


Surface Color

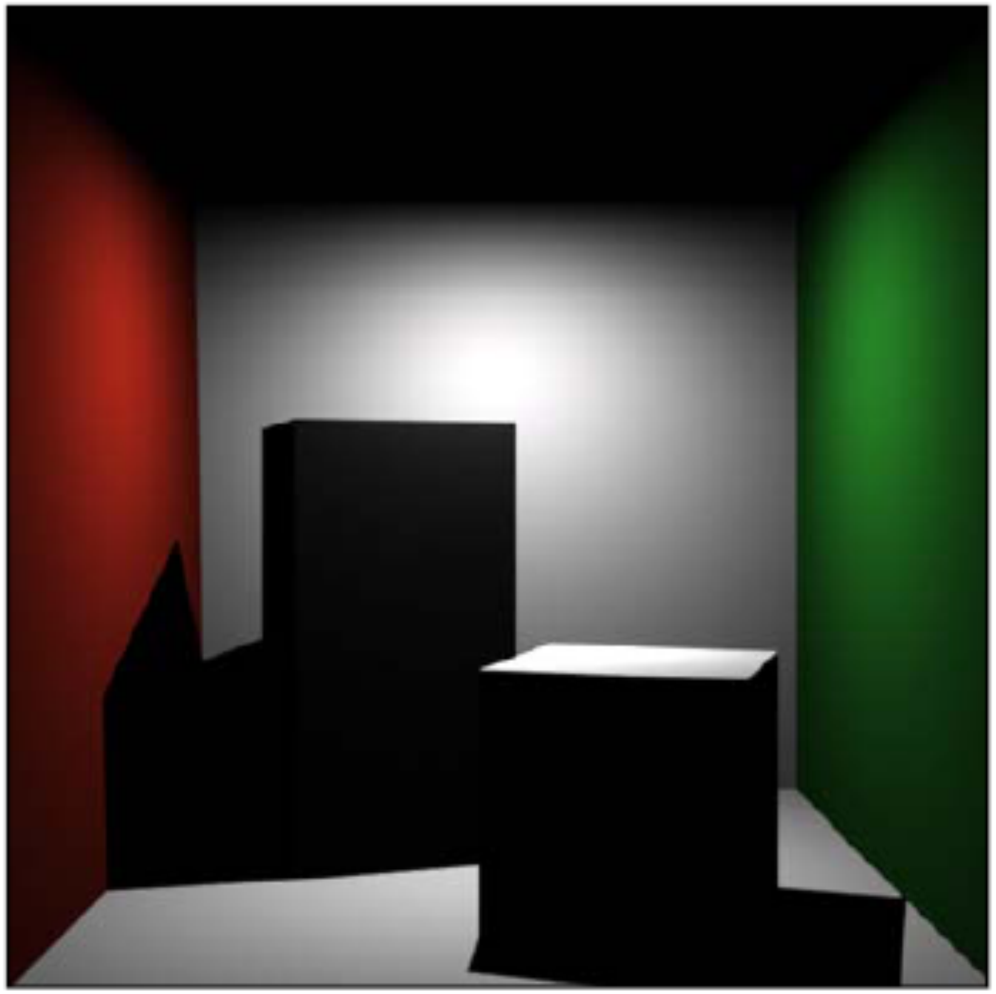


Diffuse Shading
Point Light Source

Lighting: Shadows

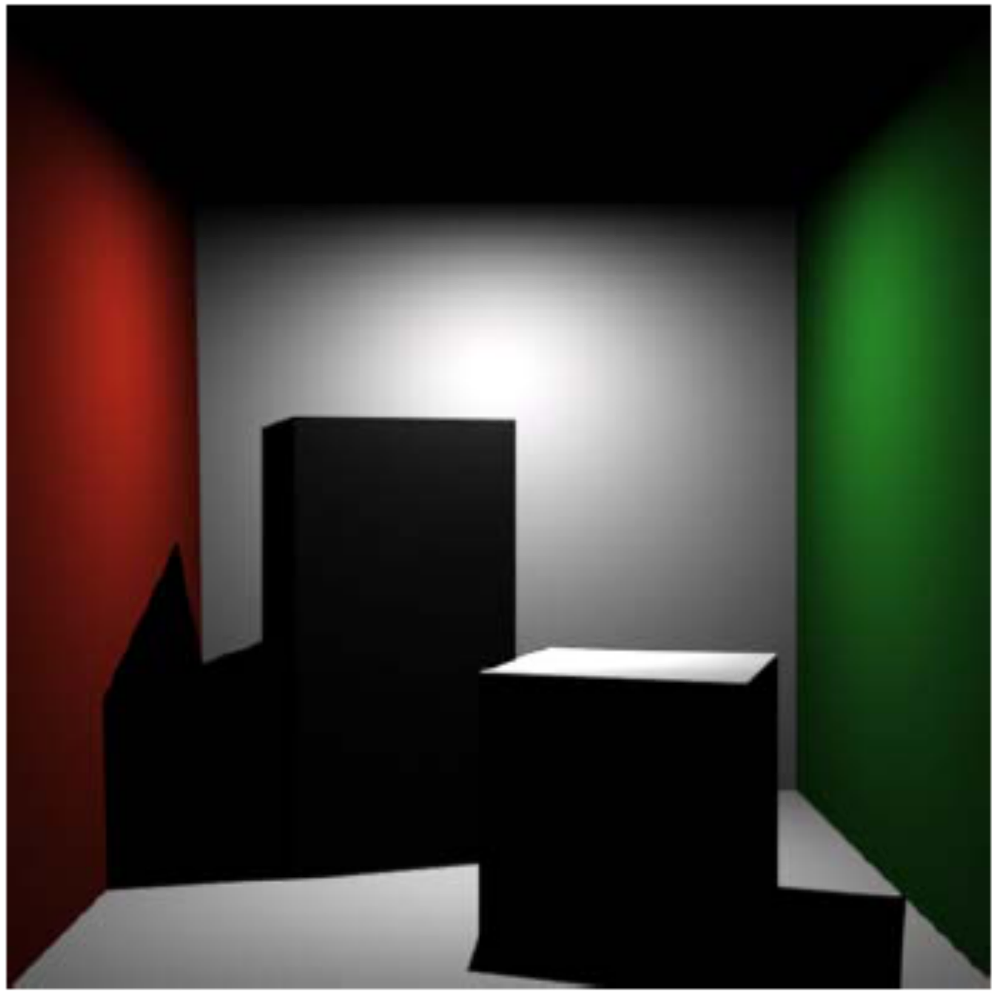


No Shadows
Point Light Source

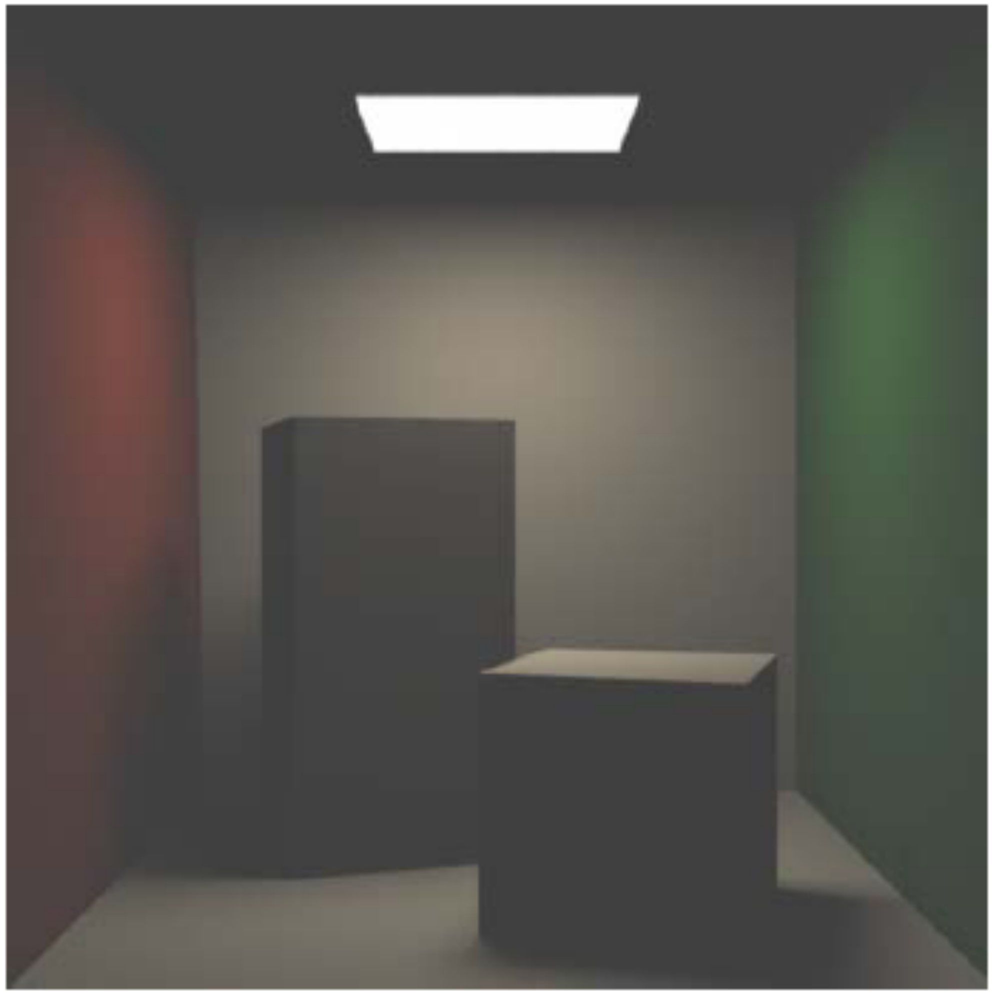


Shadows
Point Light Source

Lighting: Soft Shadows

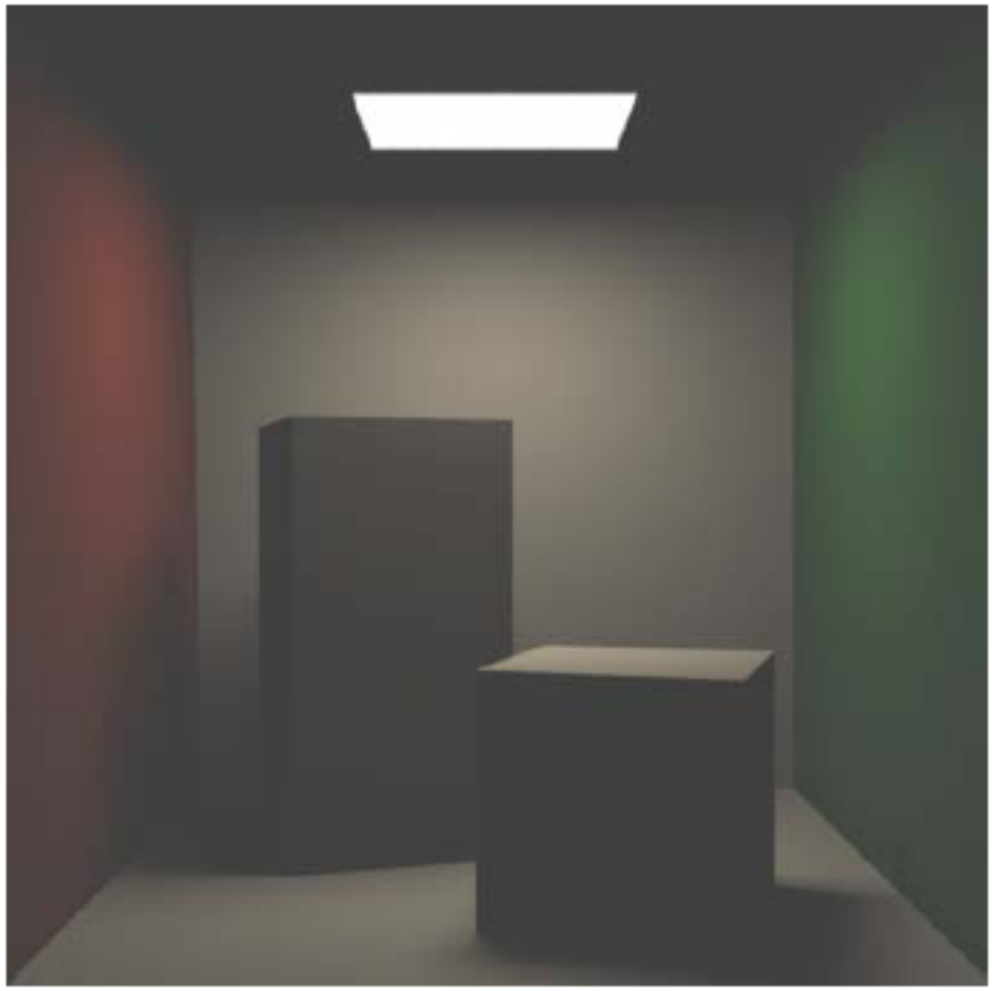


Hard Shadows
Point Light Source

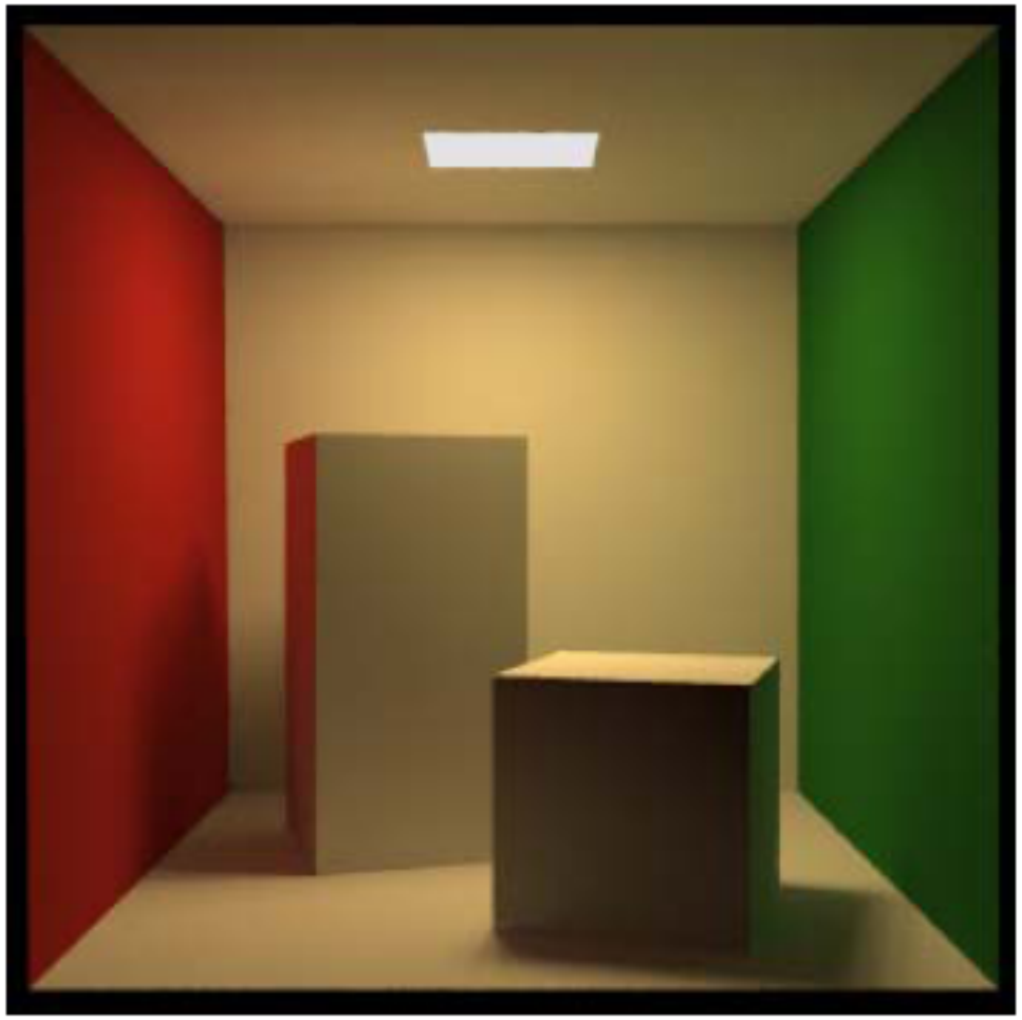


Soft Shadows
Area Light Source

Lighting: Radiosity



Soft Shadows
Area Light Source



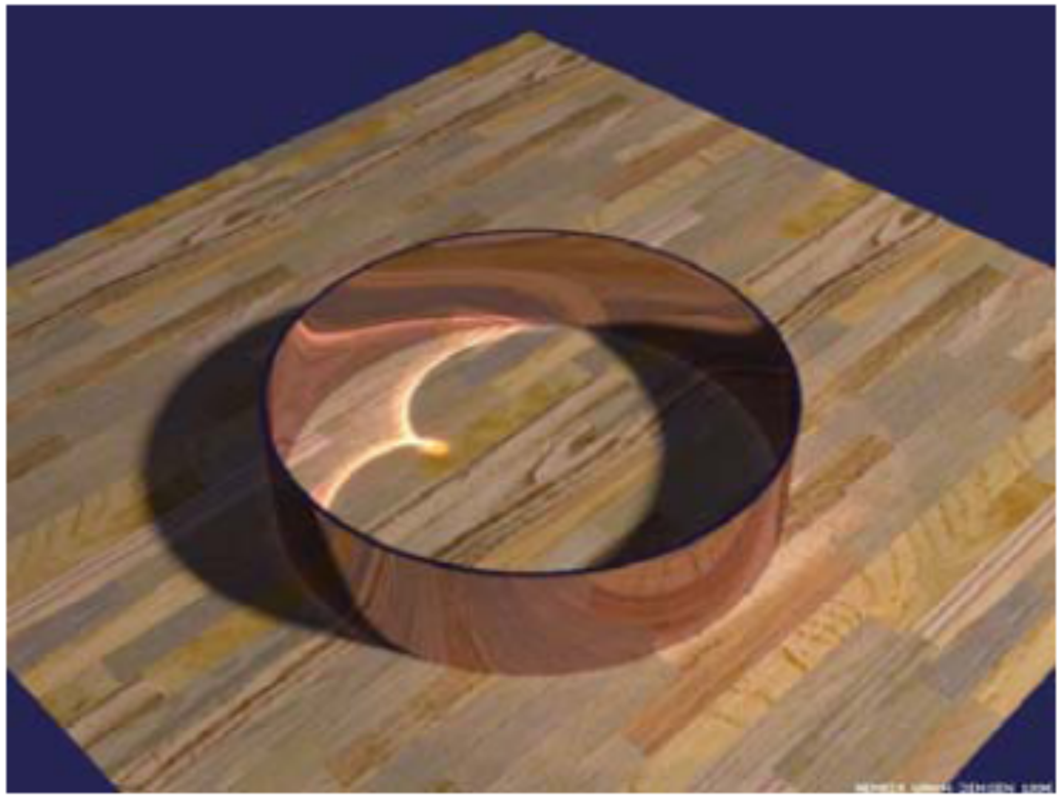
Inter-reflection, Diffuse)
Area Light Source

Early Radiosity

Early Diffuse+Glossy

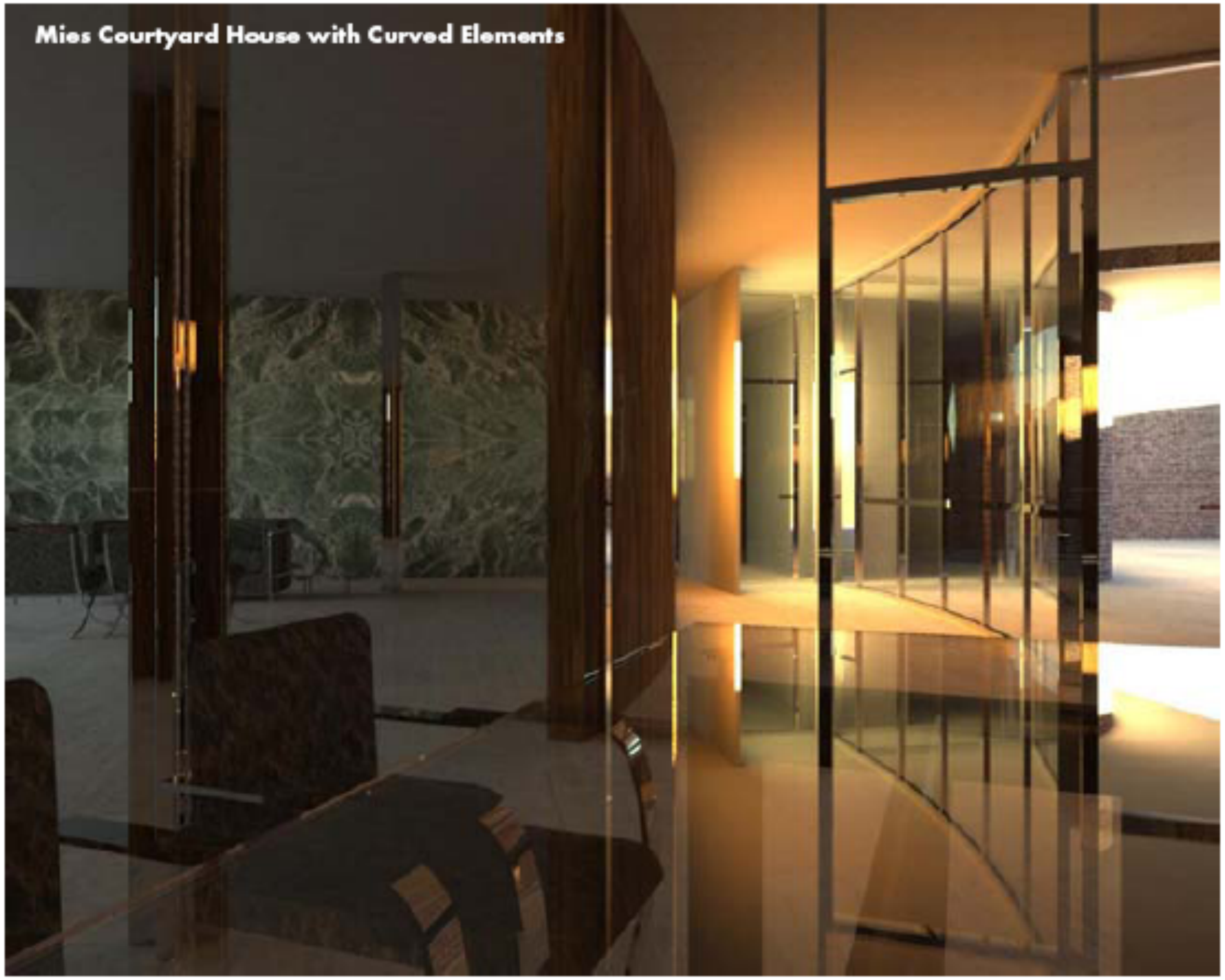
Tribute to Vermeer
Program of Computer Graphics, Cornell

Caustics



Jensen 1995

Complex Indirect Illumination



Modeling: Stephen Duck; Rendering: Henrik Wann Jensen

Translucency

Surface Reflection **Subsurface Reflection**

Rendering Algorithms

SIGGRAPH2005

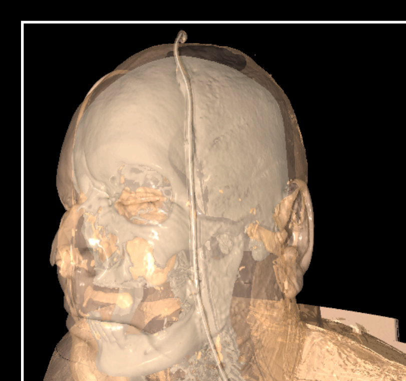
Introduction to Realtime Ray Tracing

Course 41

Philipp Slusallek Peter Shirley

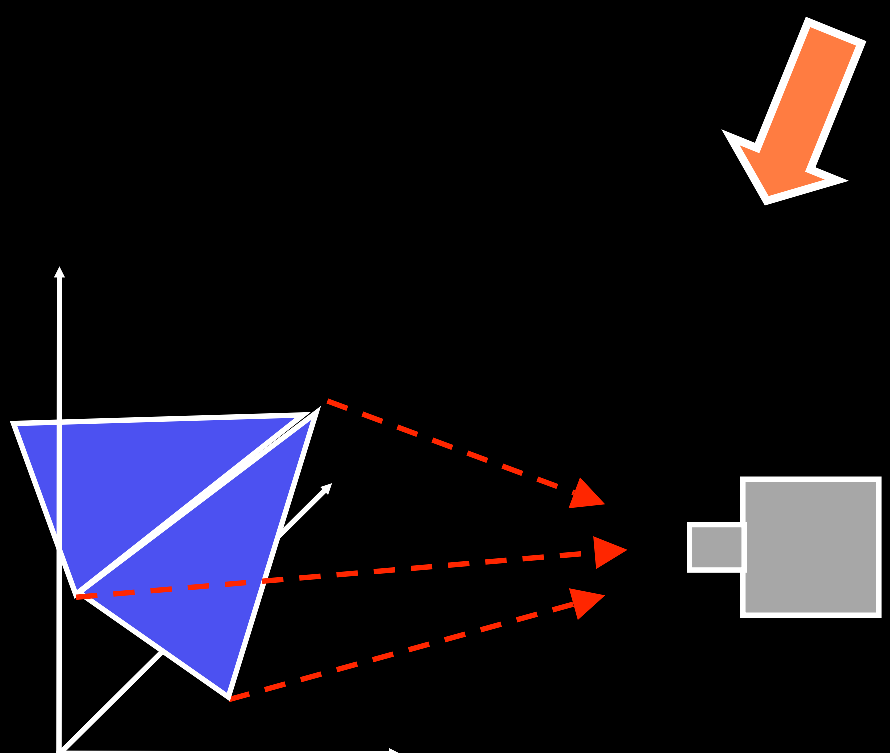
Bill Mark Gordon Stoll Ingo Wald



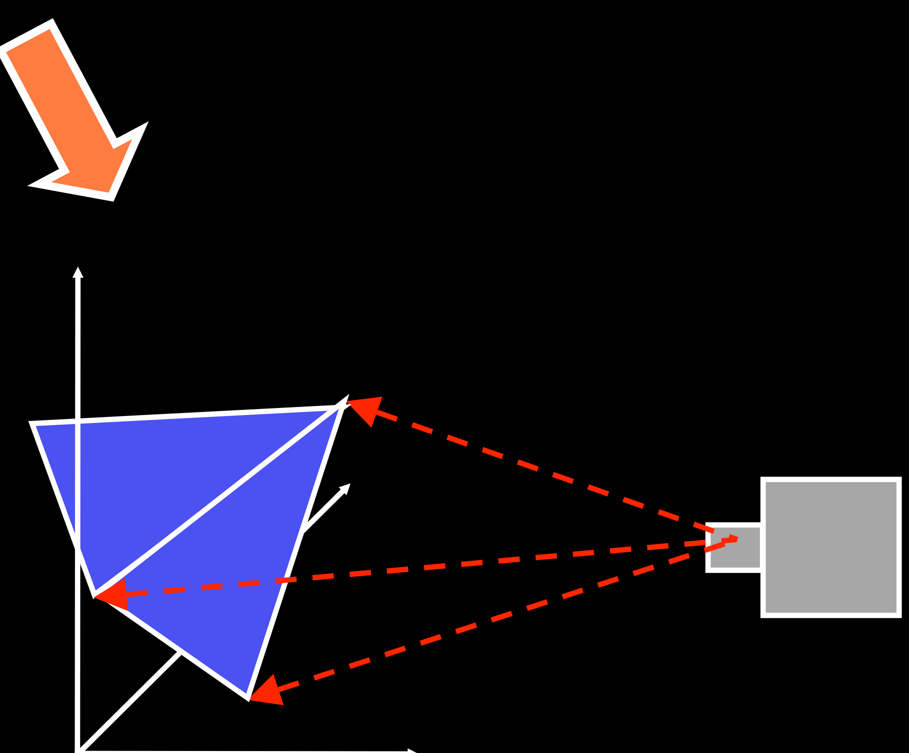



Rendering Algorithms

Rendering in Computer Graphics



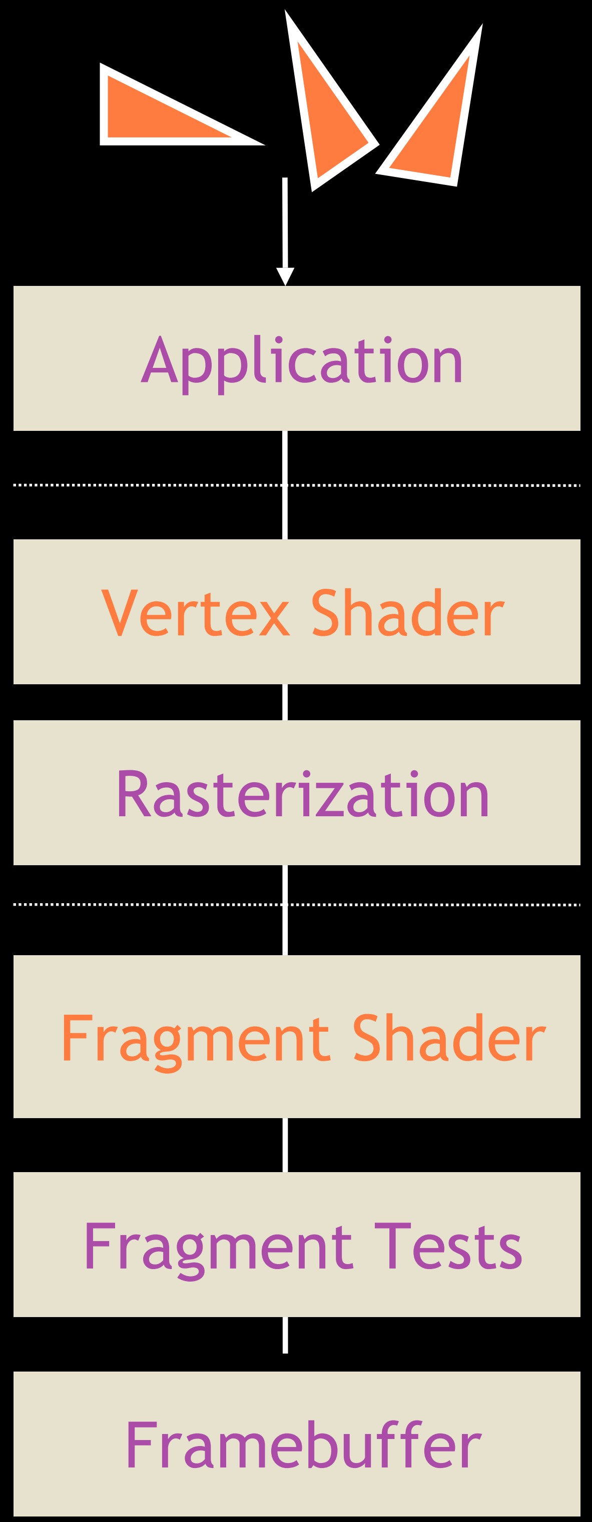
Rasterization:
Projection geometry forward



Ray Tracing:
Project image samples backwards

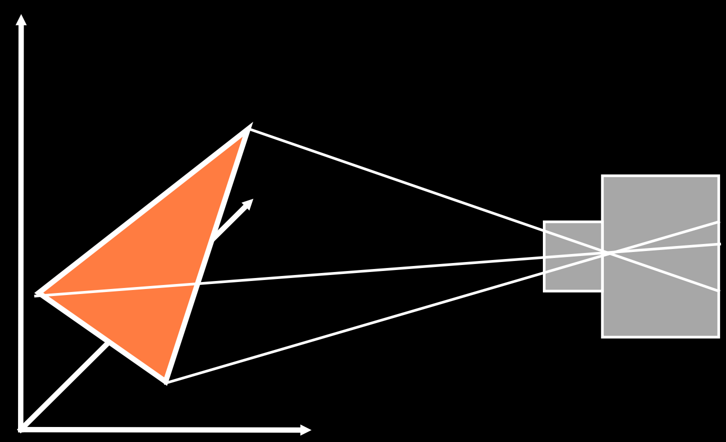
Current Technology: Rasterization

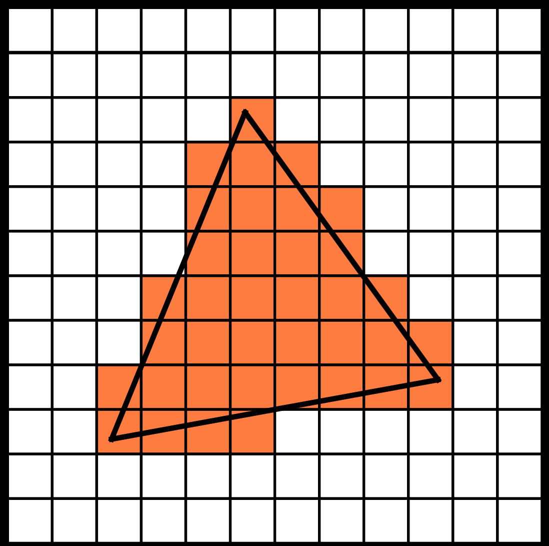
- Rasterization-Pipeline
  - Highly successful technology
  - From graphics supercomputers to an add-on in a PC chip-set
- Advantages
  - Simple and proven algorithm
  - Getting faster quickly
  - Trend towards full programmability



Current Technology: Rasterization

SIGGRAPH2005

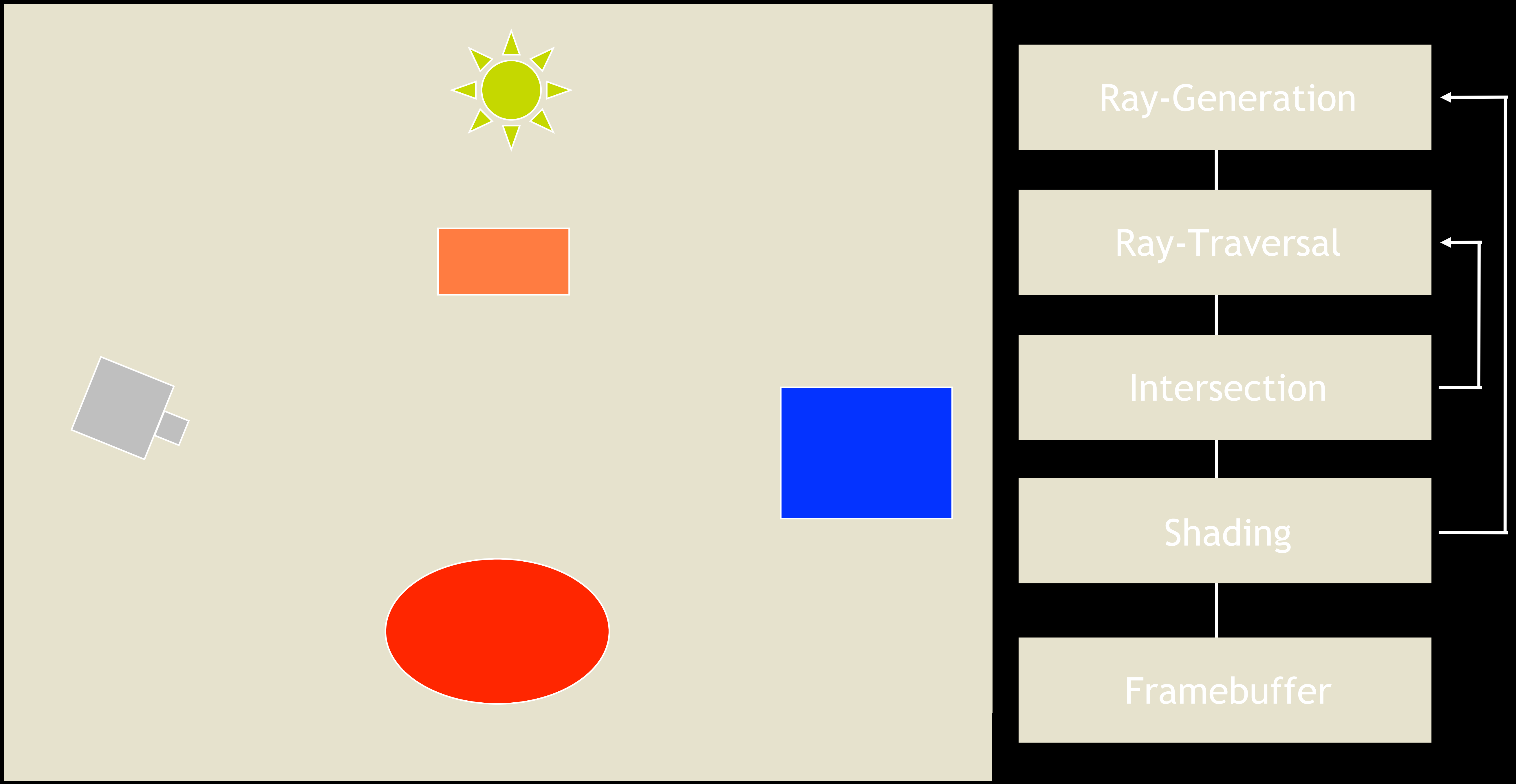




- Primitive operation of all interactive graphics !!
  - Scan converts a single triangle at a time
- Sequentially processes *every* triangle *individually*
  - Cannot access more than one triangle at a time

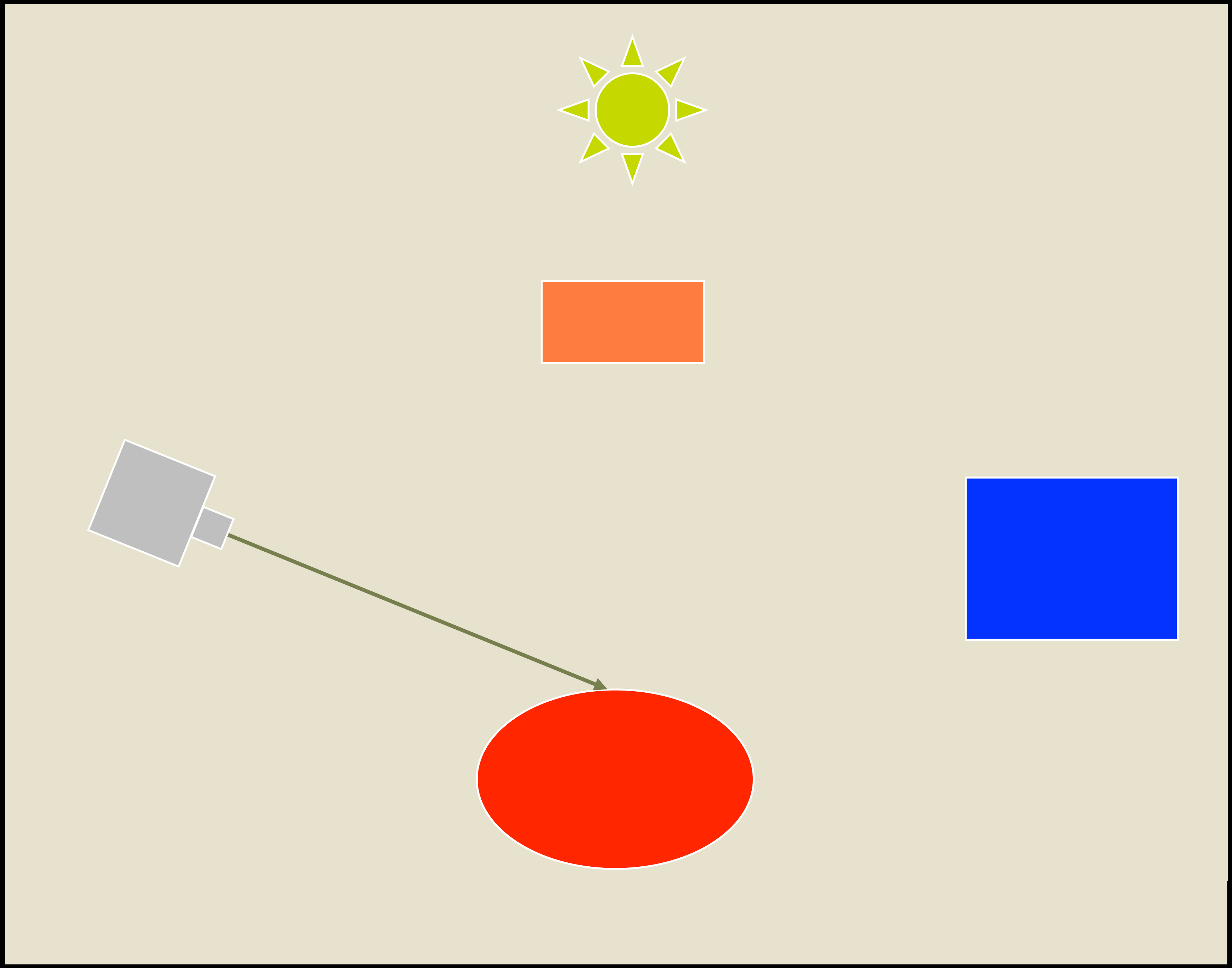
➔ But most effects need access to the entire scene:
Shadows, reflection, global illumination

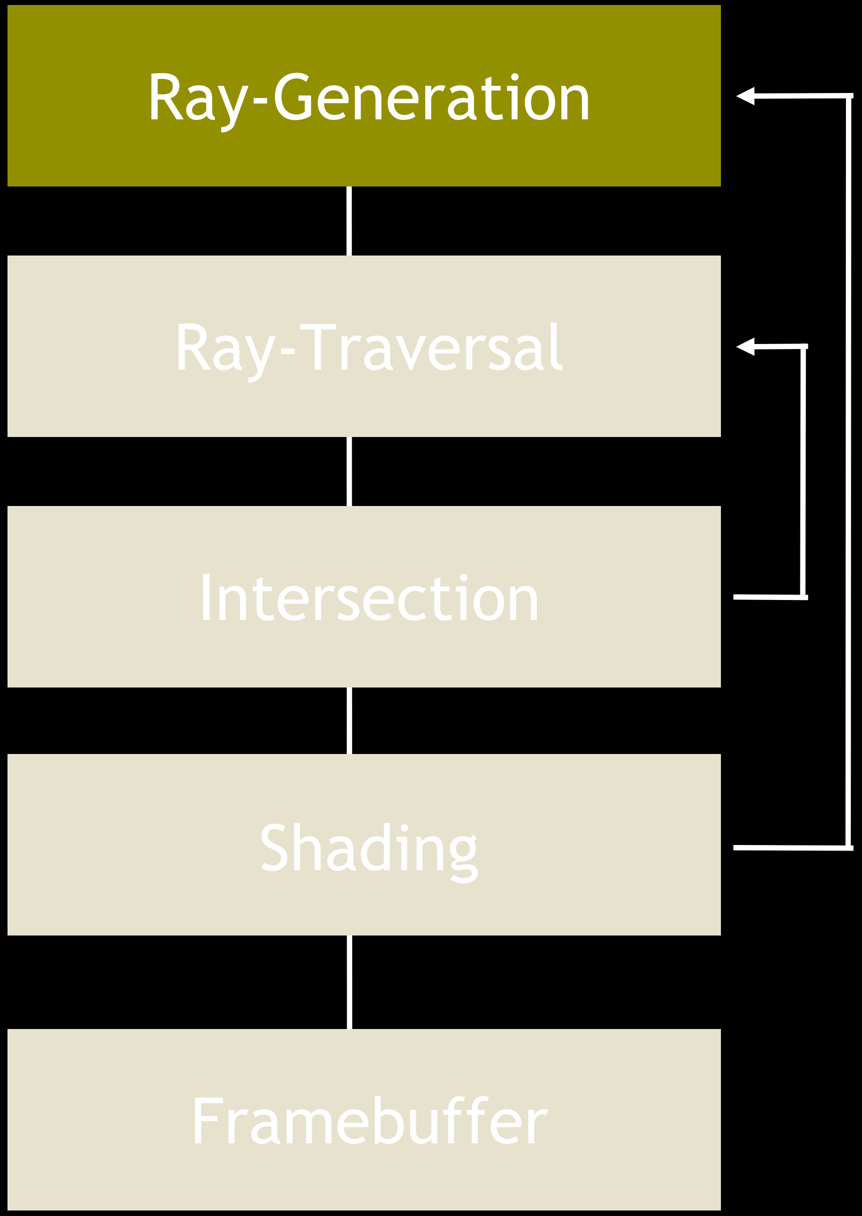
What is Ray Tracing?



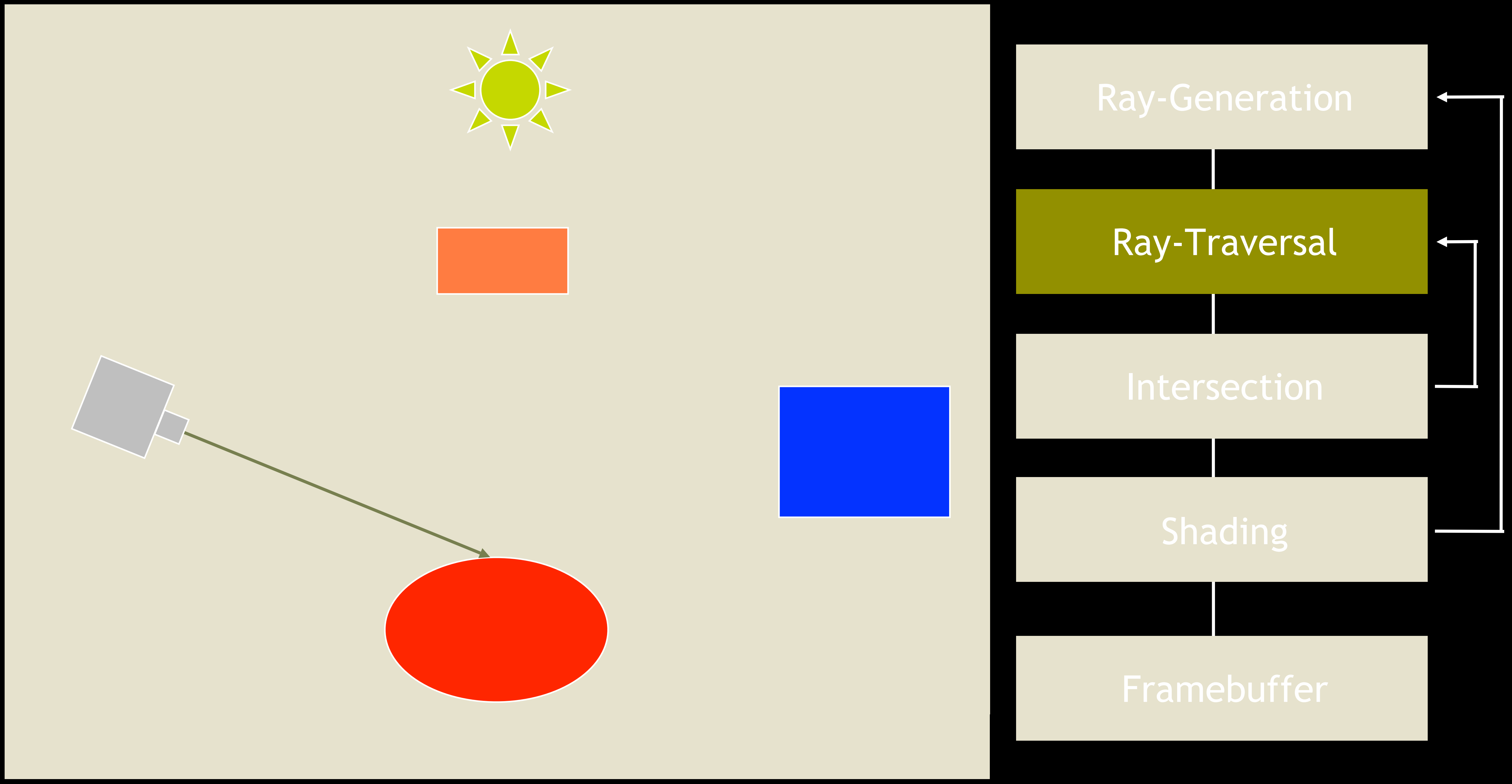
What is Ray Tracing?

SIGGRAPH2005



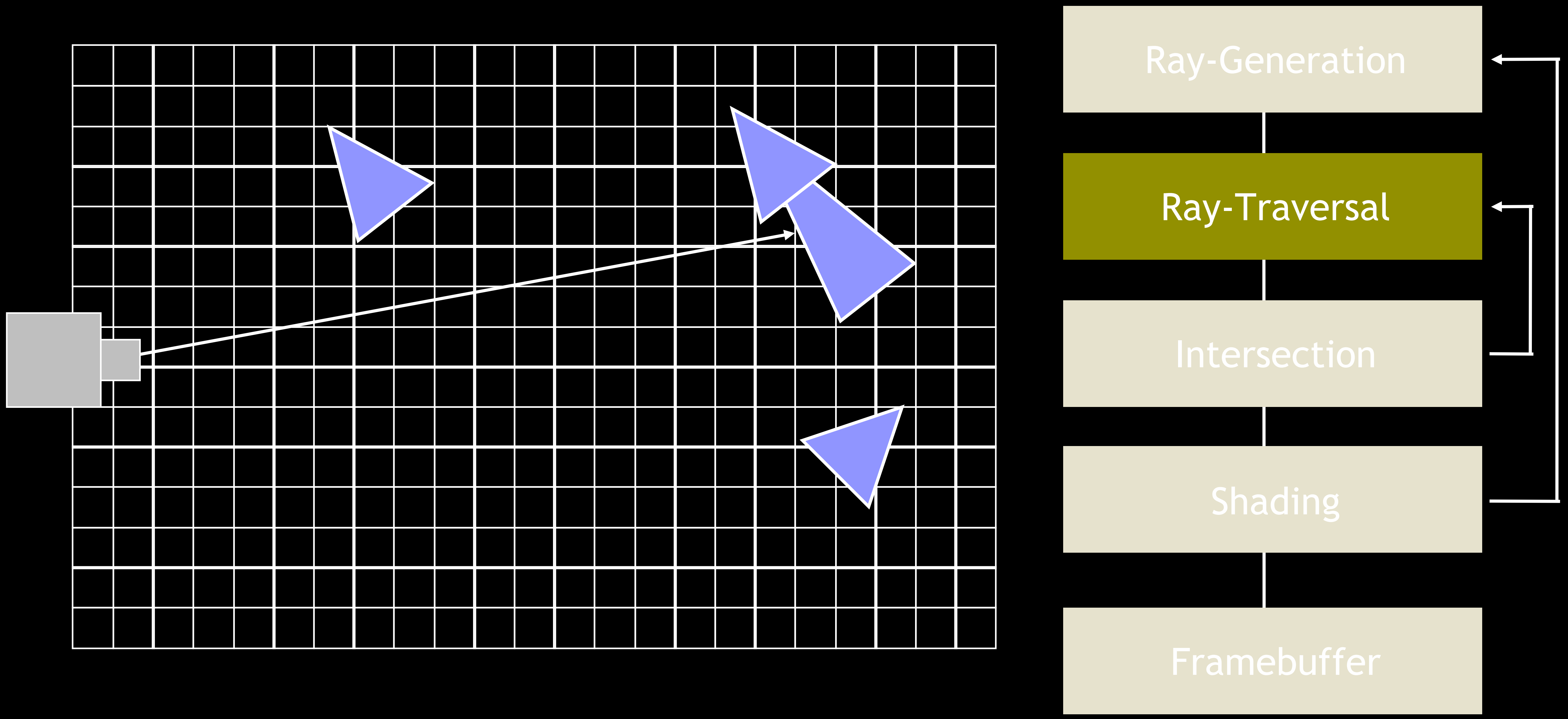


What is Ray Tracing?



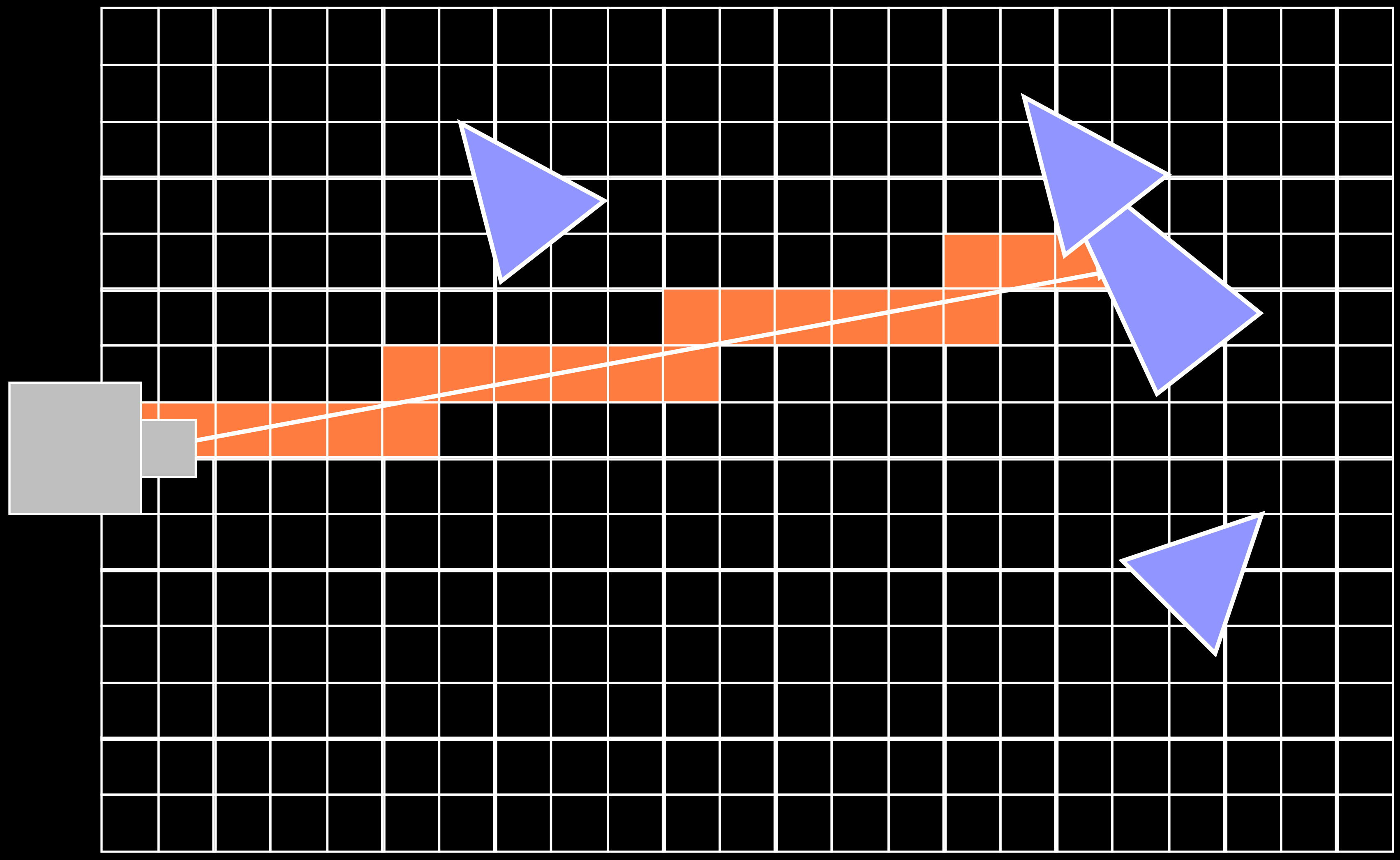
What is Ray Tracing? Traversal

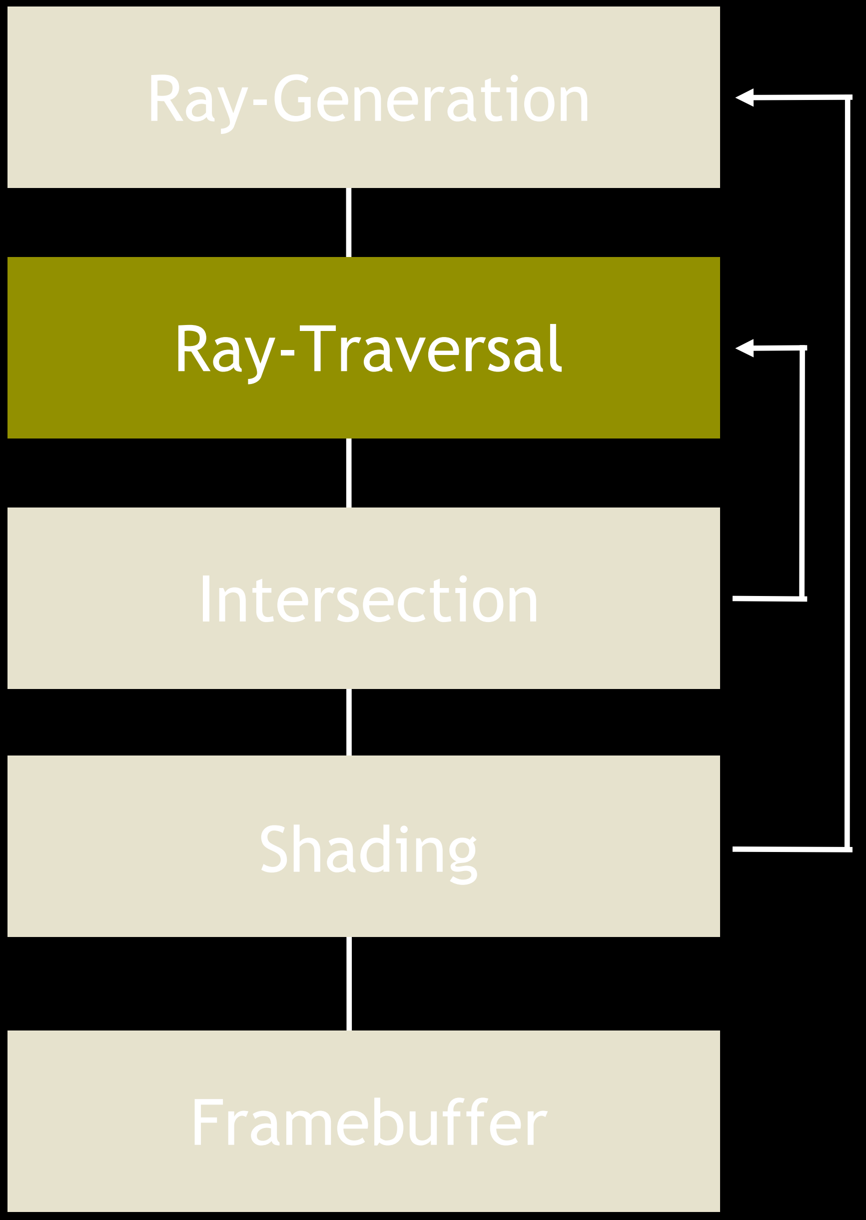
SIGGRAPH2005



What is Ray Tracing? Traversal

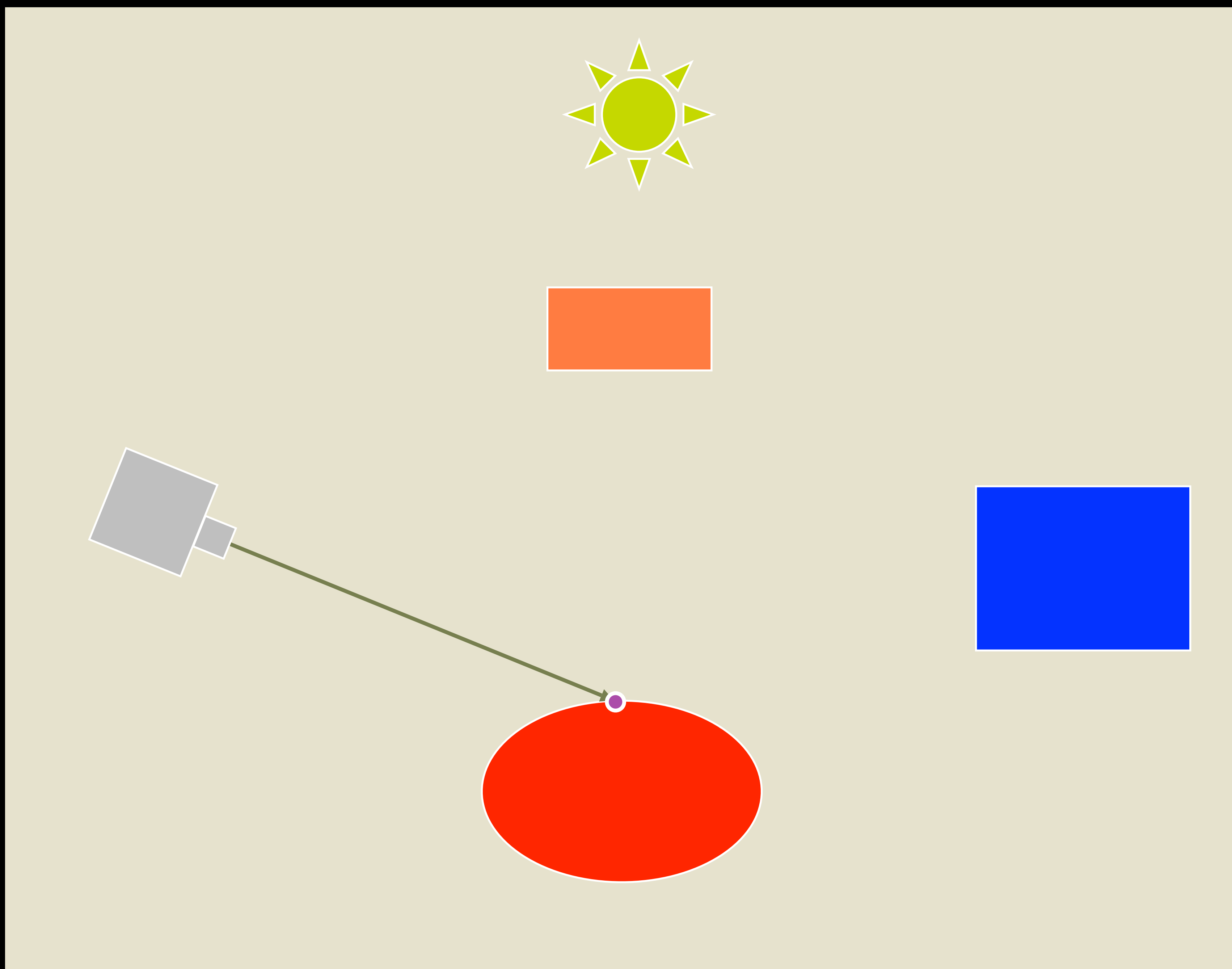
SIGGRAPH2005

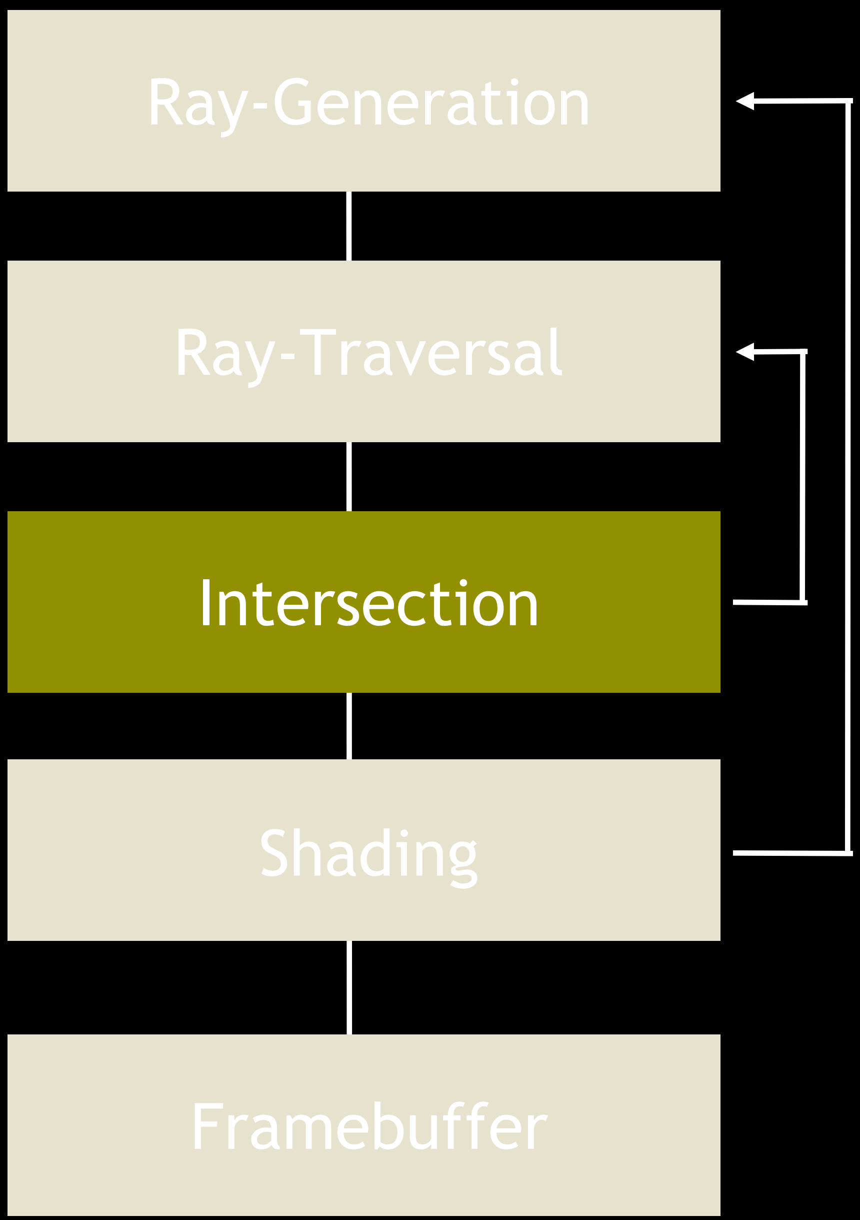




What is Ray Tracing?

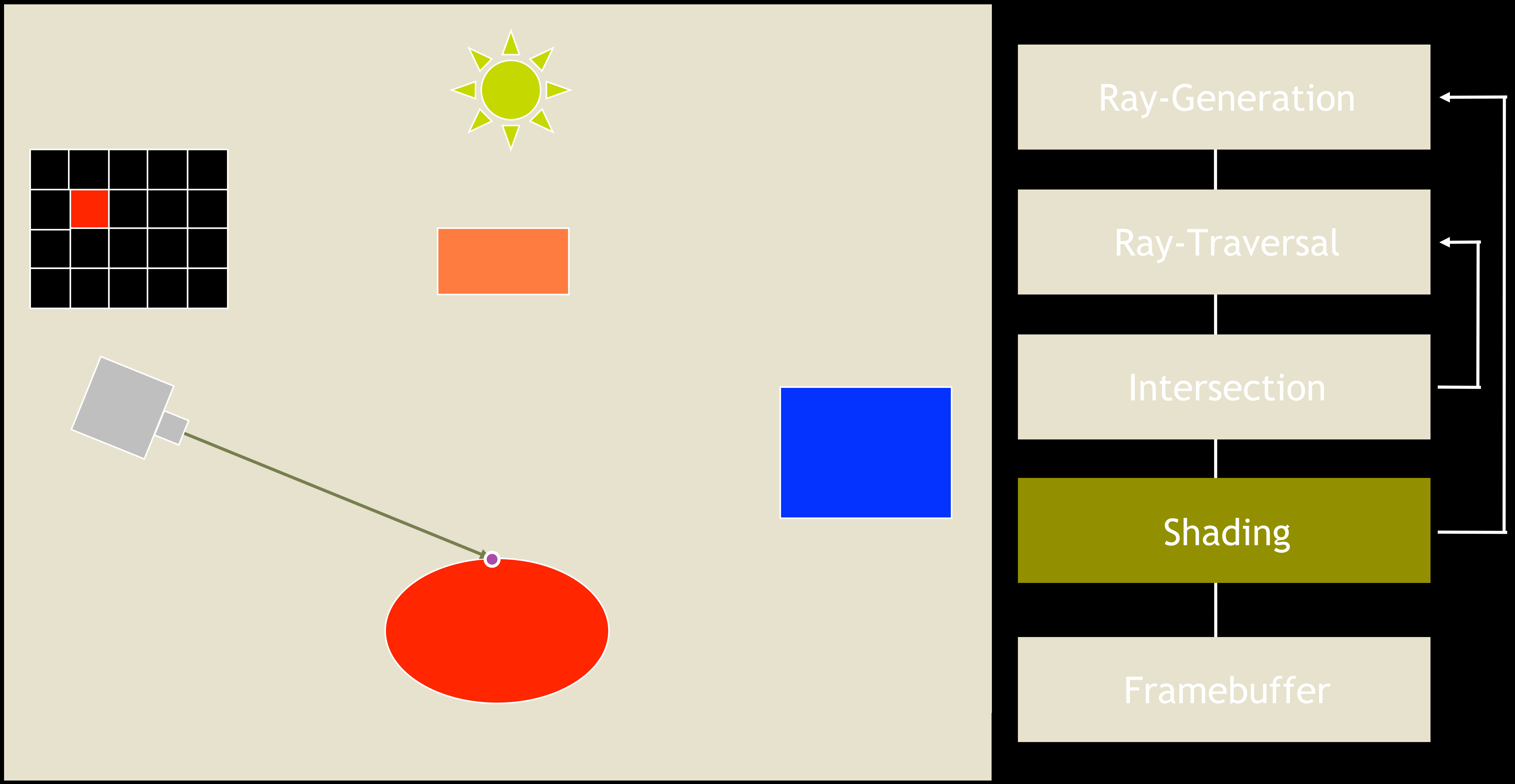
SIGGRAPH2005



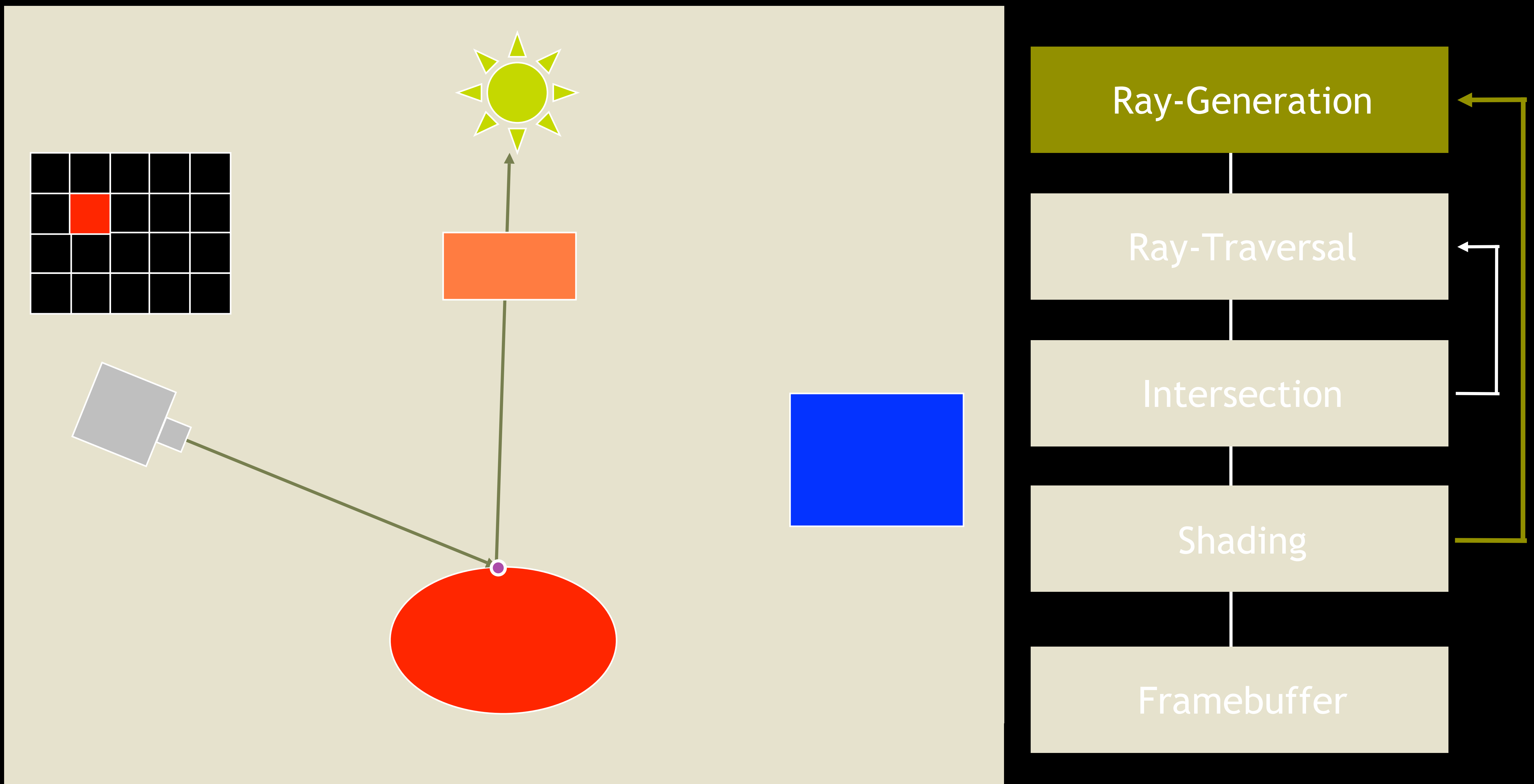


What is Ray Tracing?

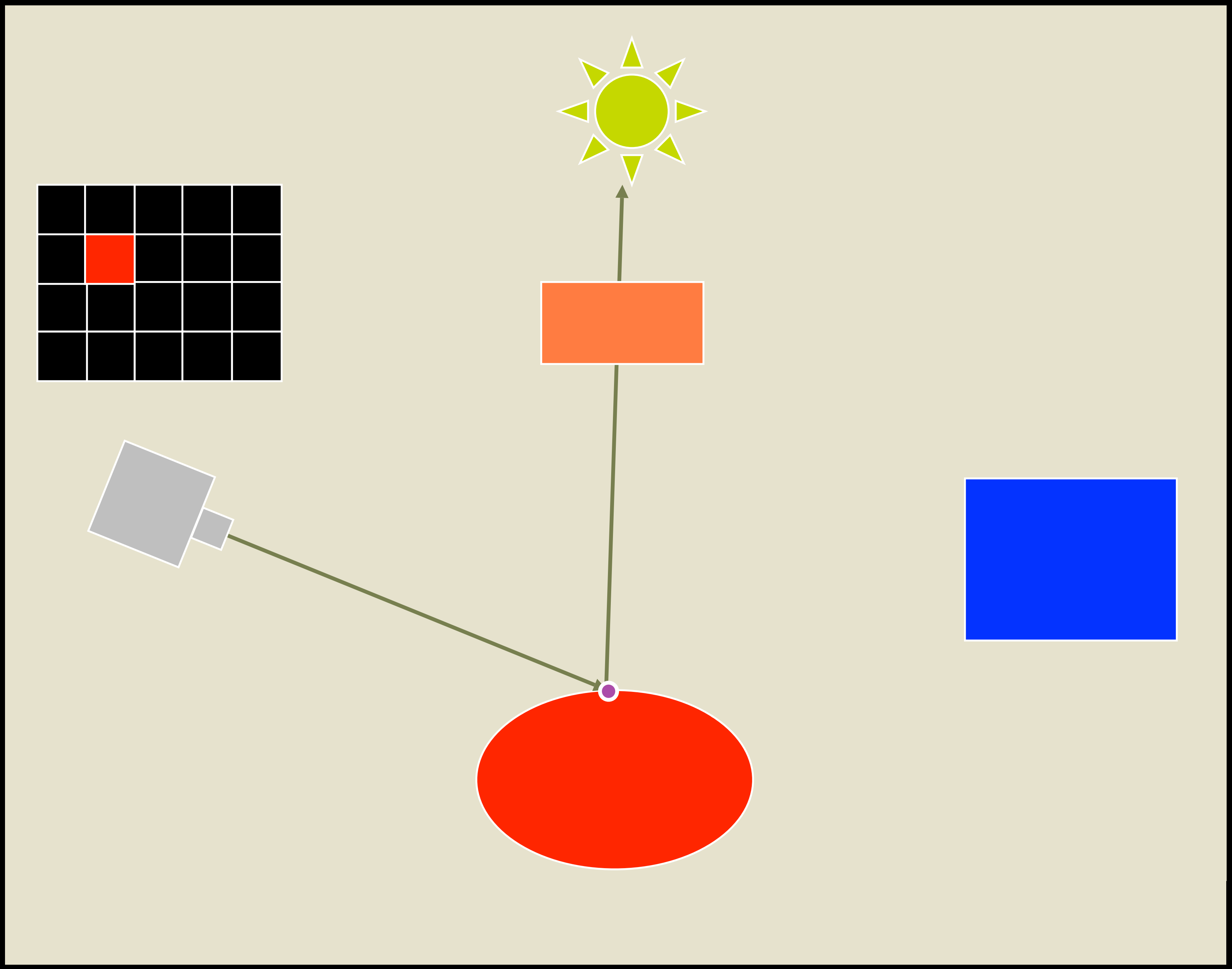
SIGGRAPH2005

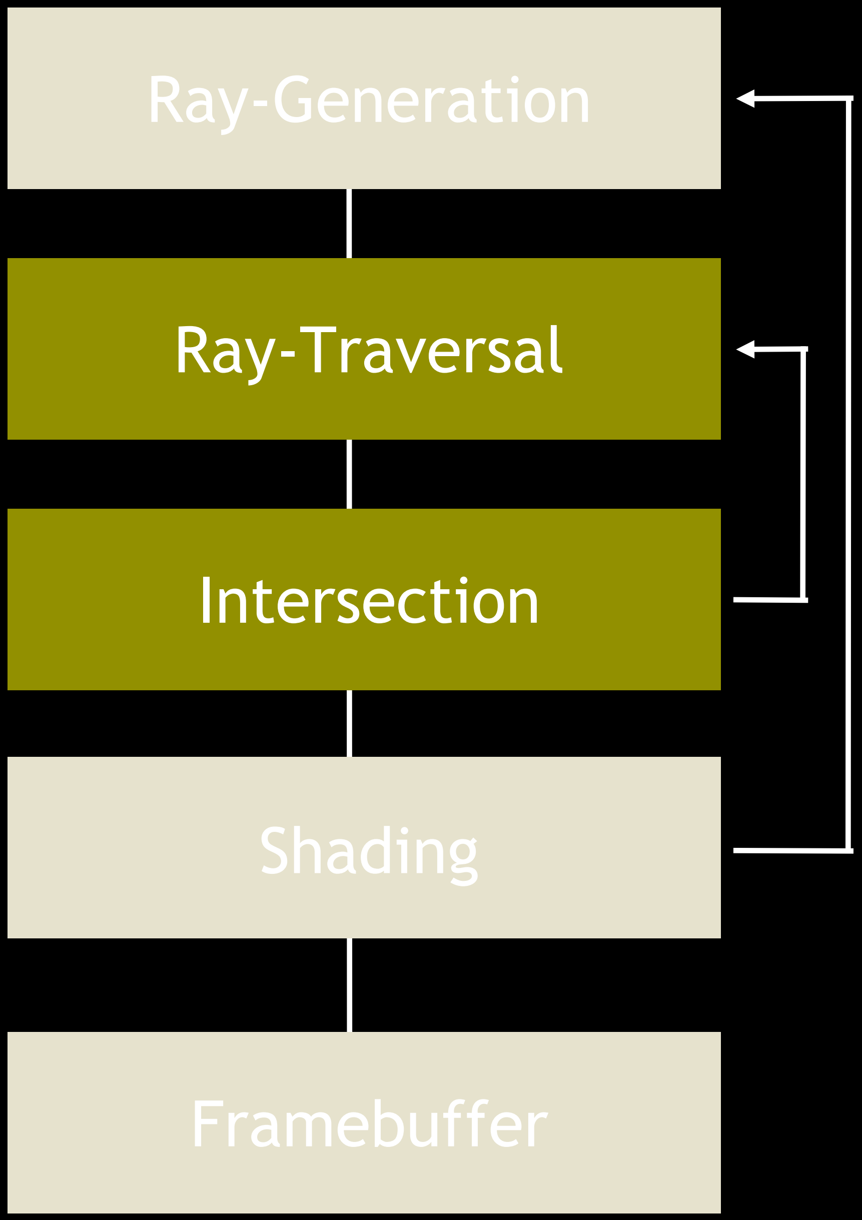


What is Ray Tracing?

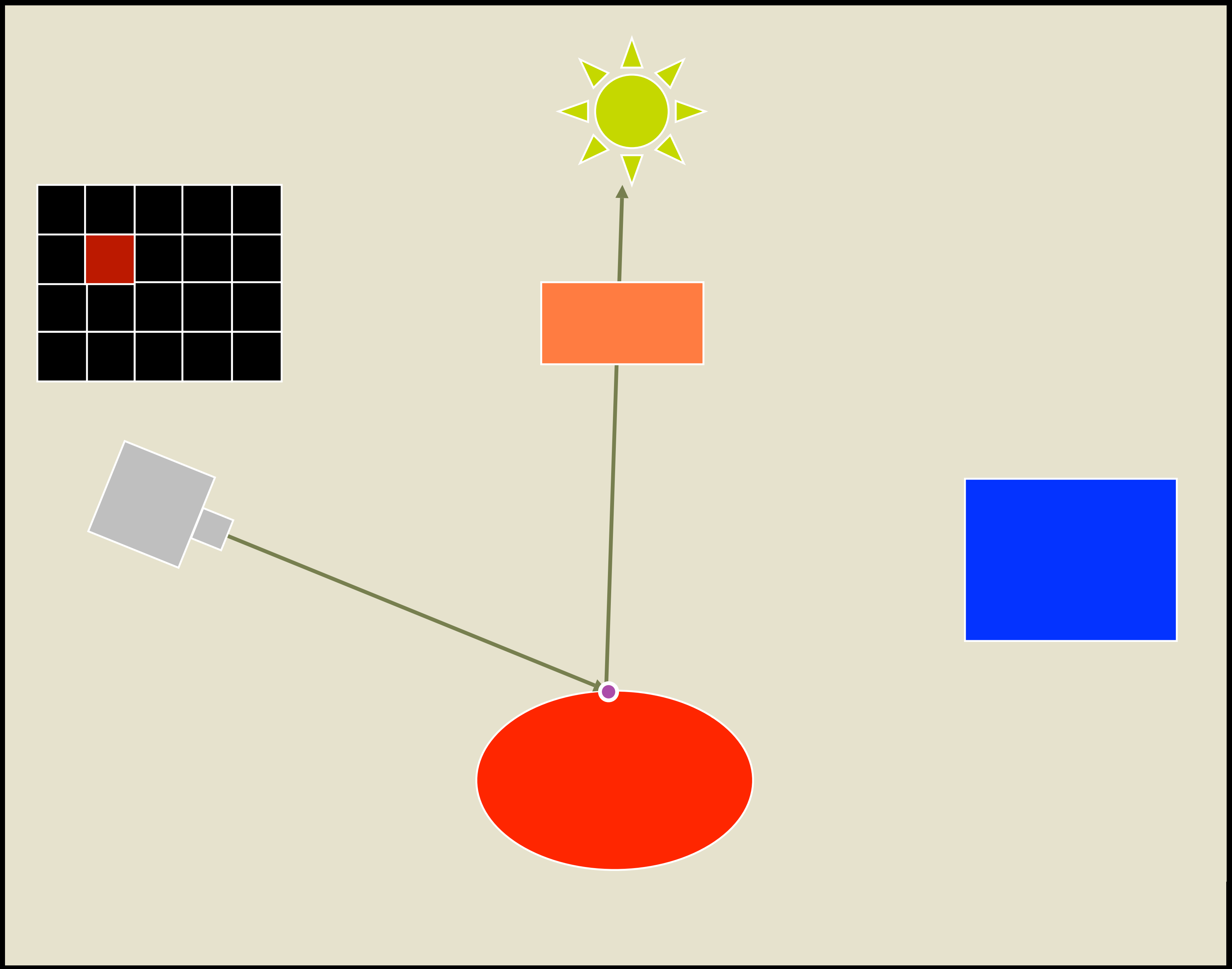


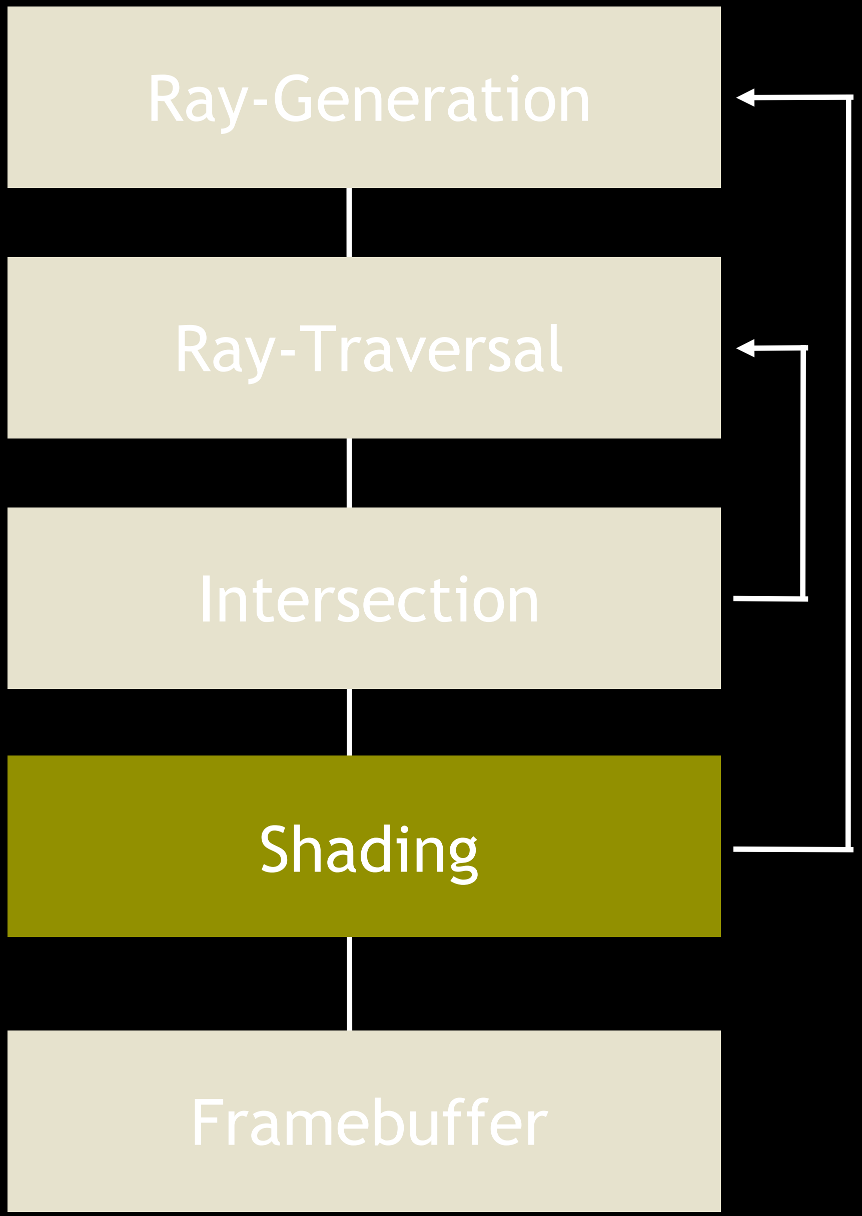
What is Ray Tracing?





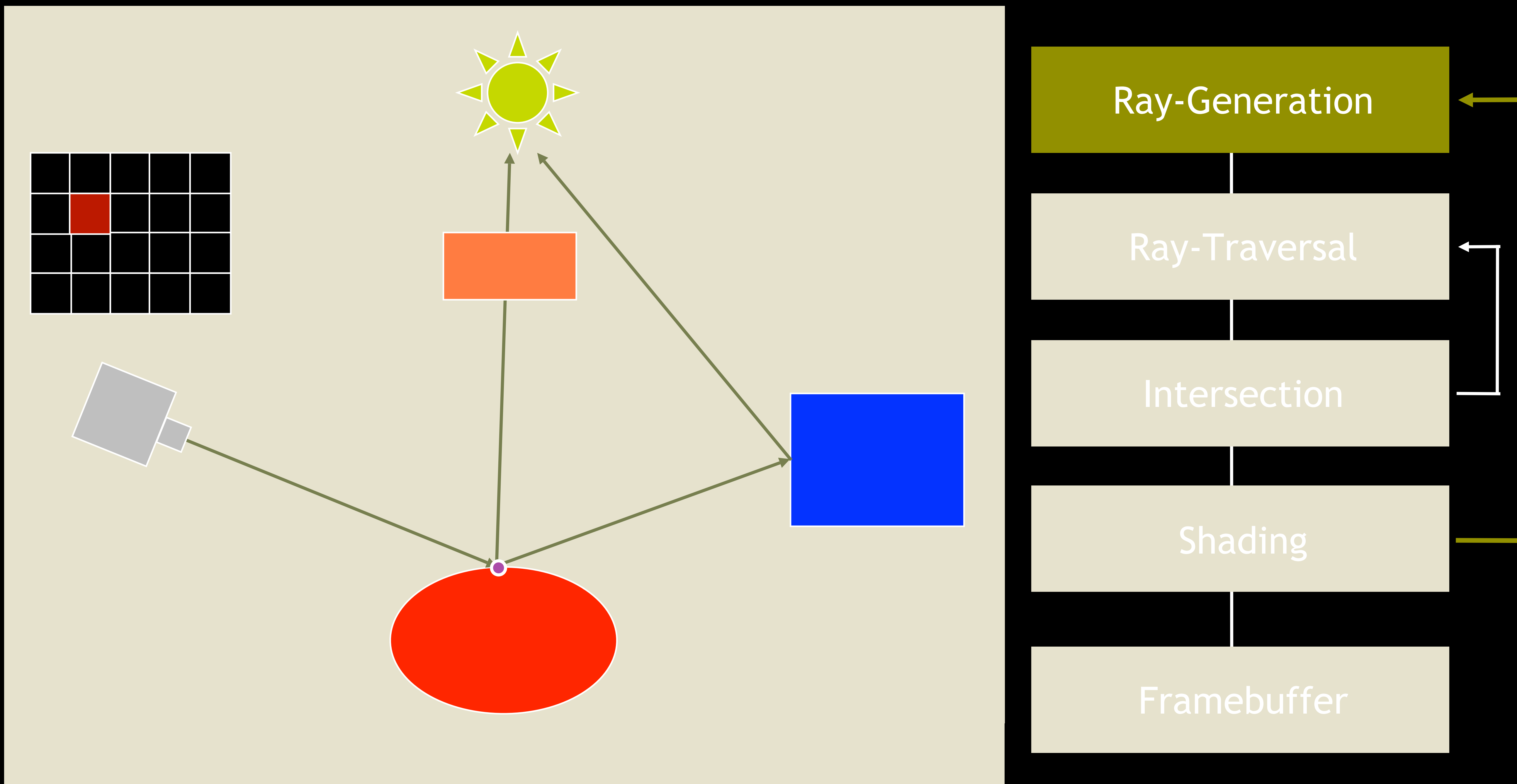
What is Ray Tracing?





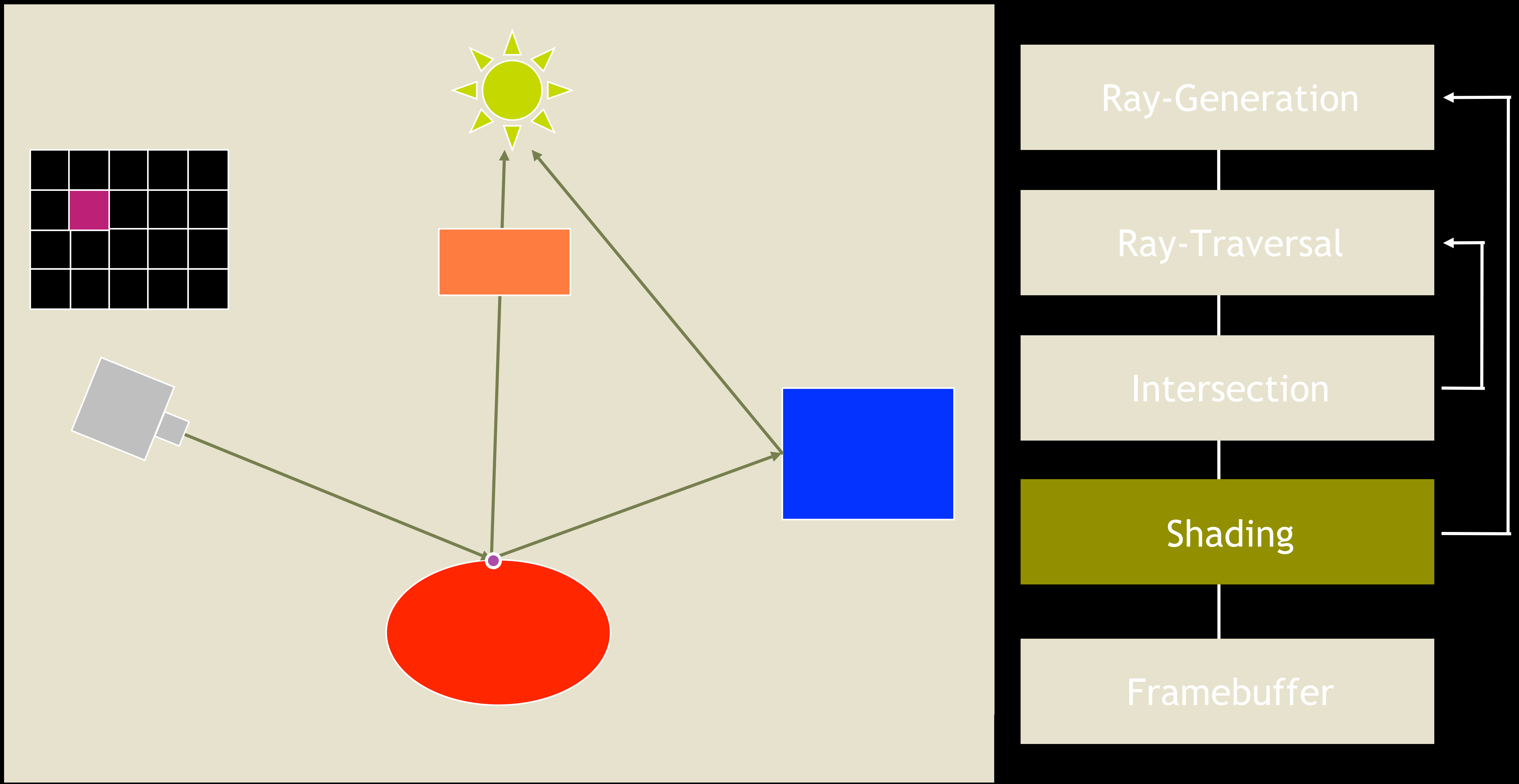
What is Ray Tracing?

SIGGRAPH2005



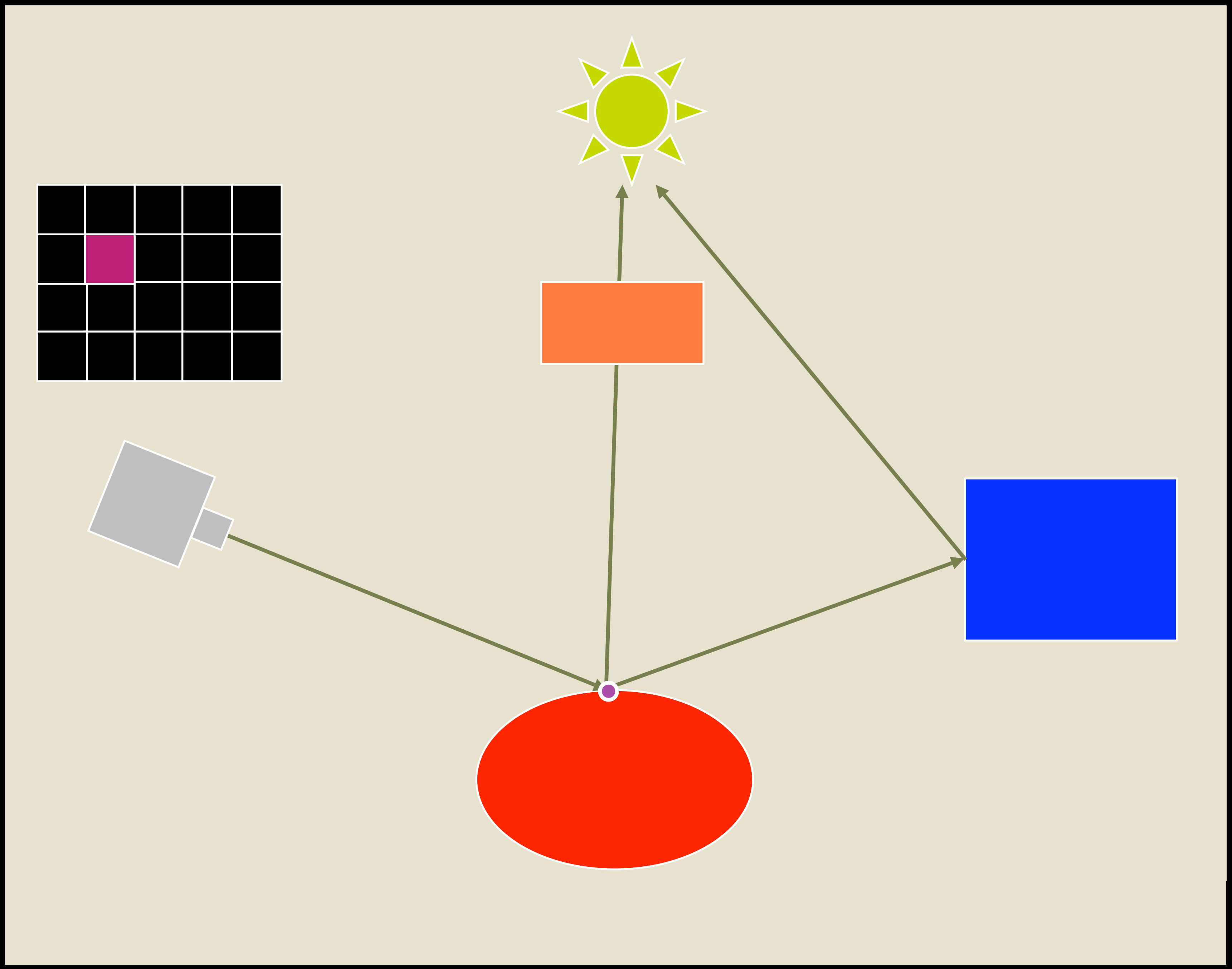
What is Ray Tracing?

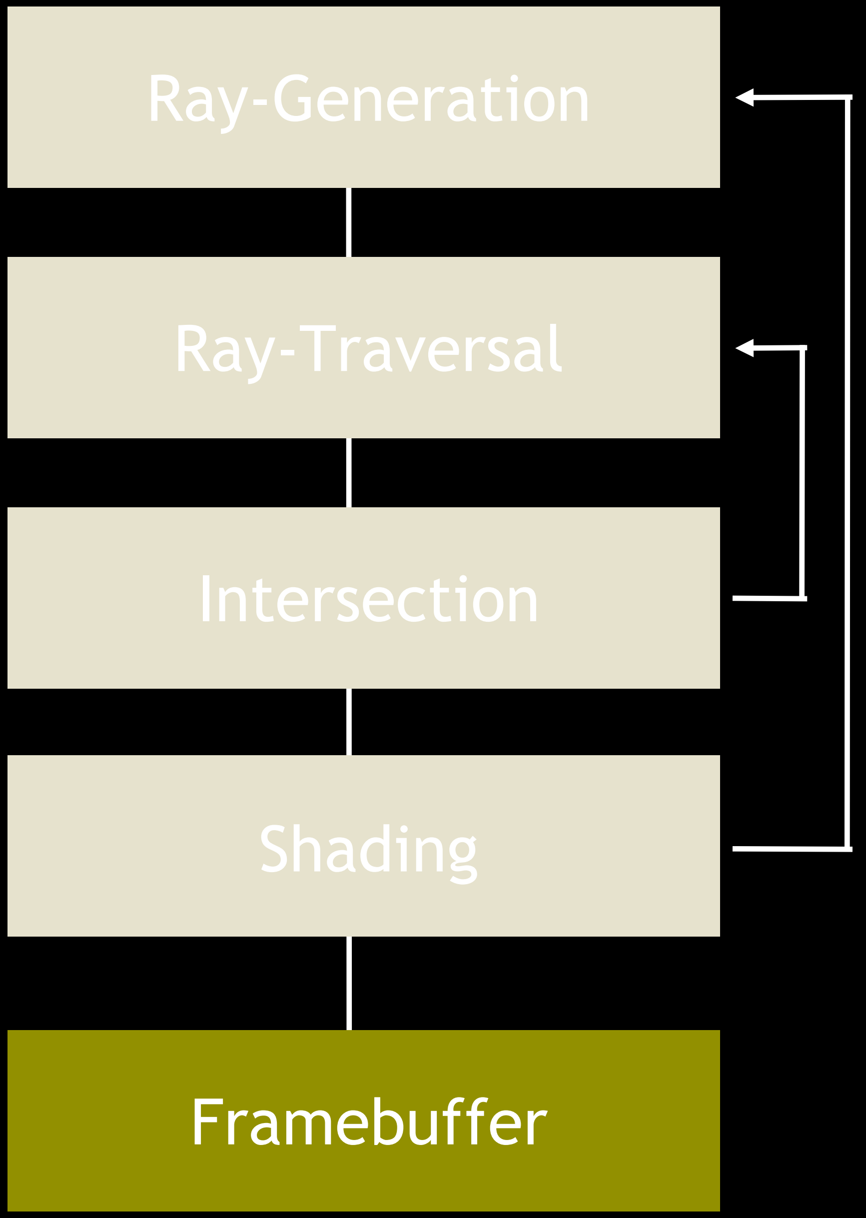
SIGGRAPH2005



What is Ray Tracing?

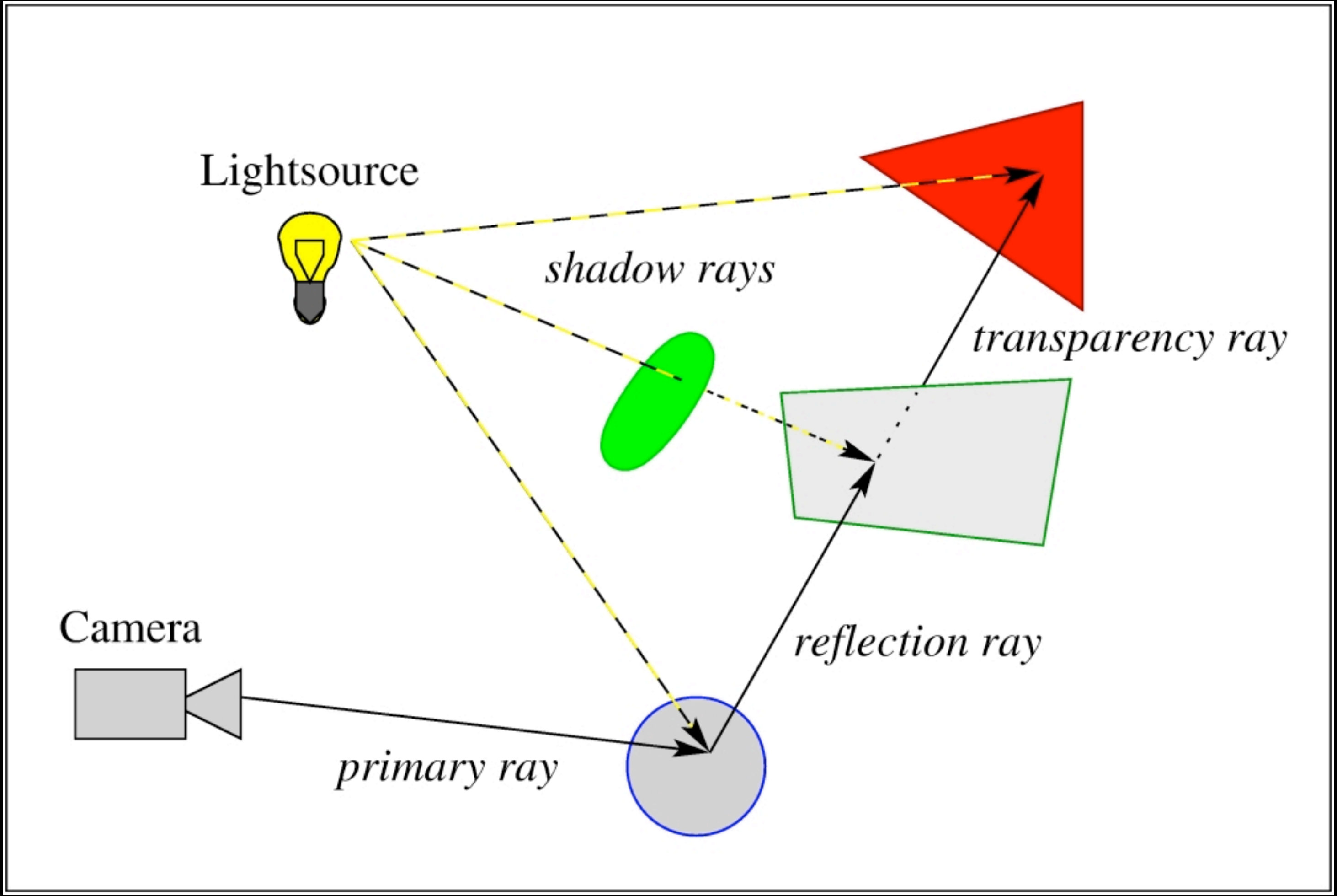
SIGGRAPH2005





What is Ray Tracing?

SIGGRAPH2005



- Global effects
- Parallel (as nature)
- Fully automatic
- Demand driven
- Per pixel operations
- Highly efficient

→ Fundamental Technology for Next Generation Graphics

Comparison
Rasterization vs. Ray Tracing

- Definition: Rasterization

  Given a set of rays and a primitive, efficiently compute the subset of rays hitting the primitive

- Definition: Ray Tracing

  Given a ray and set of primitives, efficiently compute the subset of primitives hit by the ray

Comparison
Rasterization vs. Ray Tracing

- Hardware Support
  - Rasterization has mature & quickly evolving HW
    - High-performance, highly parallel, stream computing engine
  - Ray tracing mostly implemented in SW
    - Requires flexible control flow, recursion & stacks, flexible i/o, …
    - Requires virtual memory and demand loading due scene size
    - Requires loops in the HW pipeline (e.g. generating new rays)
    - Depend heavily on caching and suitable working sets

➔ Not well supported by current HW

Reasons for Using Ray Tracing

SIGGRAPH2005

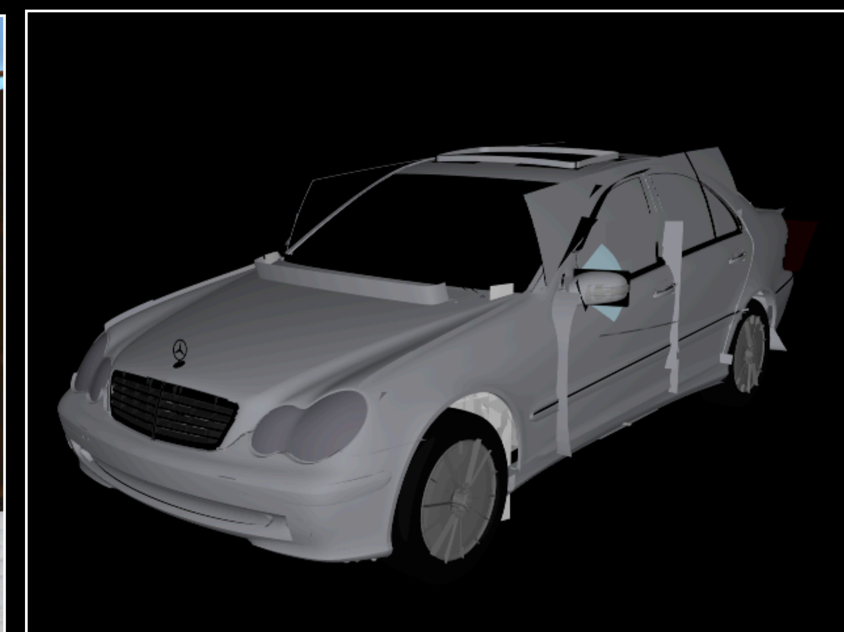
- Physical Correctness and Dependability
  - Numerous approximations caused by rasterization
  - Might be good enough for games (but maybe not?)
  - Industry needs dependable visual results
- Benefits
  - Users develop trust in the visual results
  - Important decisions can be based on virtual models

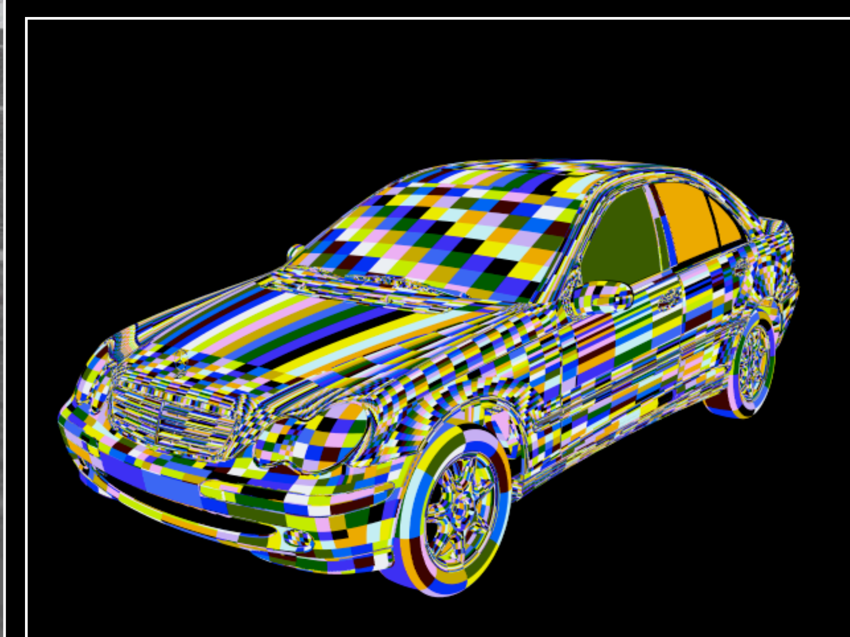
Reasons for Ray Tracing: Physical Correctness

Fully ray traced car head lamp, faithful visualization requires up to 50 rays per pixel

Reasons for Ray Tracing: Physical Correctness

SIGGRAPH2005



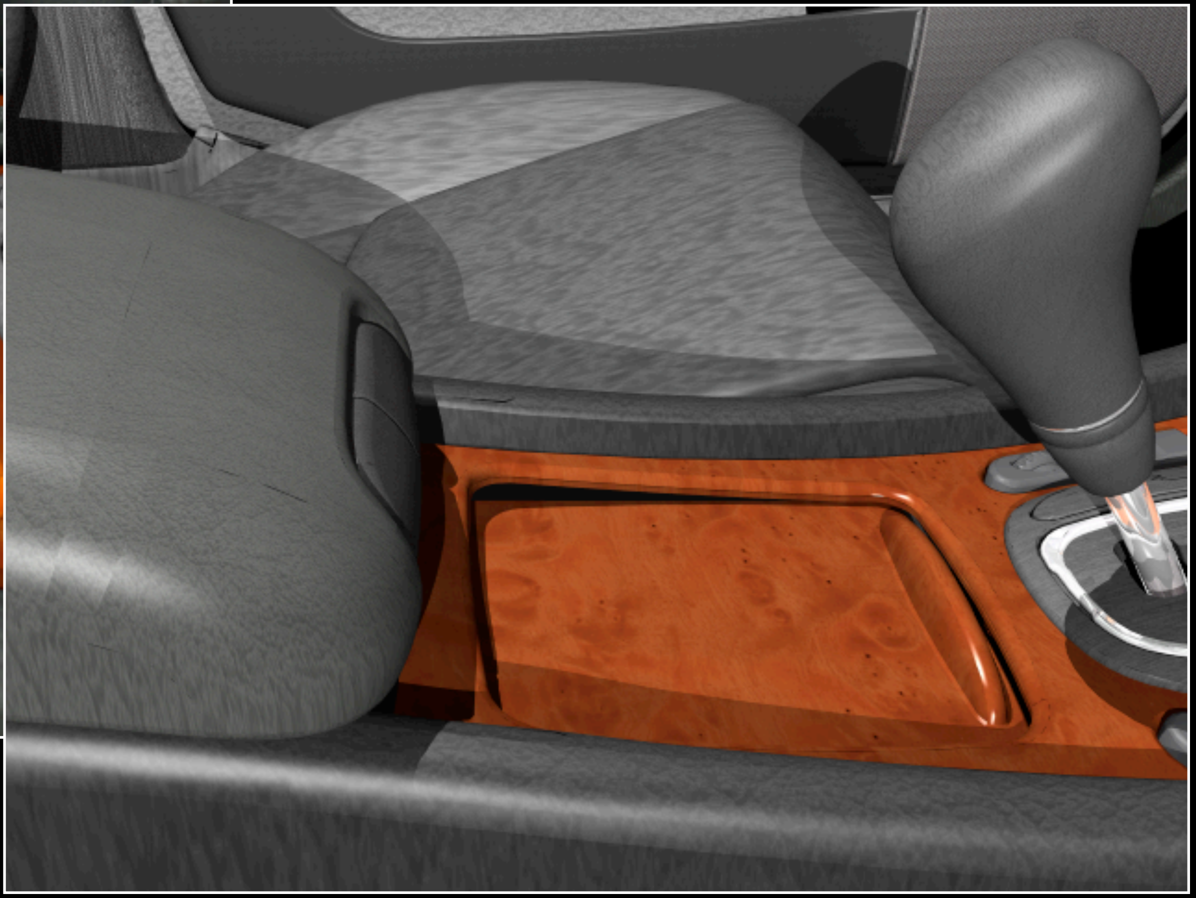


Rendered directly from trimmed NURBS surfaces, with smooth environment lighting

Reasons for Ray Tracing: Physical Correctness

SIGGRAPH2005

Textured Phong for comparison



Rendered with accurately measured BTF data that accounts for micro lighting effects

BTF Data Courtesy R. Klein, Uni Bonn

Reasons for Ray Tracing: Physical Correctness

SIGGRAPH2005

VR scene illuminated from realtime video feed, AR with realtime environment lighting

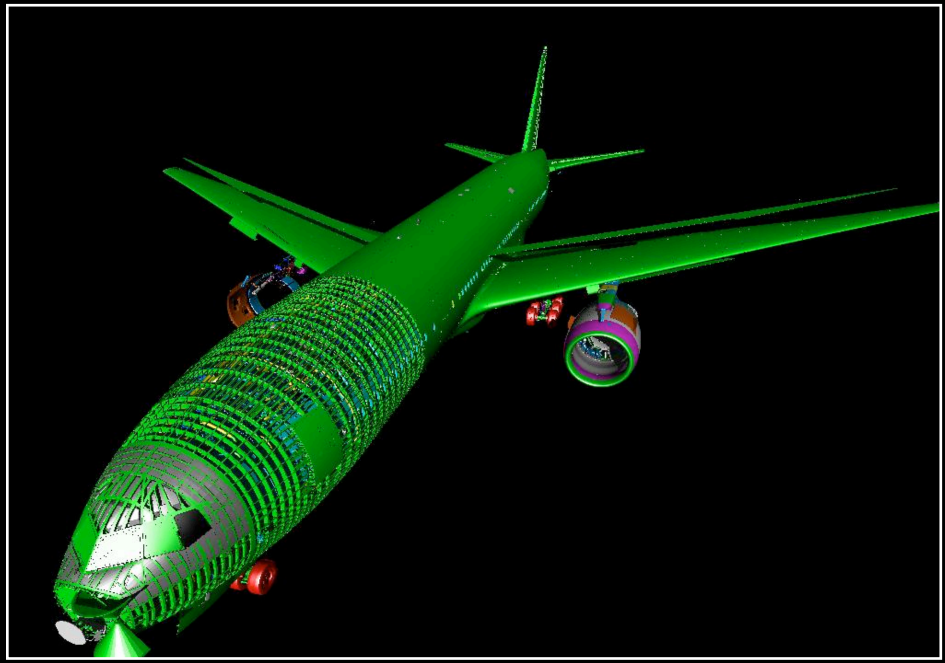
Reasons for Ray Tracing: Massive Models

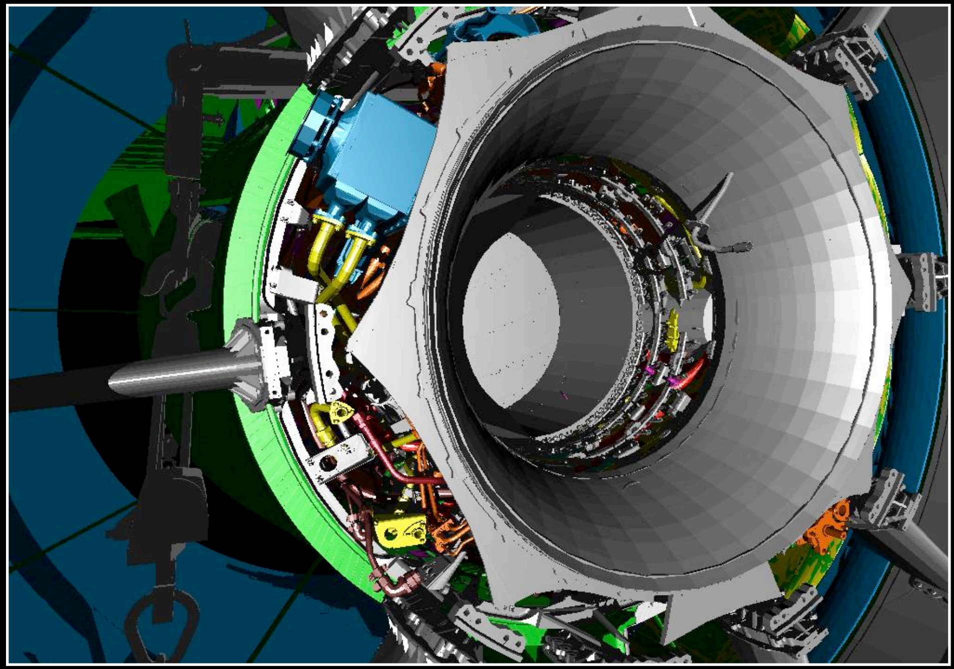
SIGGRAPH2005

- Massive Scenes
  - Scales logarithmically with scene size
  - Supports billions of triangles
- Benefits
  - Can render entire CAD models without simplification
  - Greatly simplifies and speeds up many tasks

Reasons for Ray Tracing: Massive Models

SIGGRAPH2005





Reasons for Ray Tracing: Flexible Primitive Types

- Flexible Primitive Types
  - Triangles
  - Volumes data sets
    - Iso-surfaces & direct visualization
    - Regular, rectilinear, curvilinear, unstructured, …
  - Splines and subdivision surfaces
  - Points

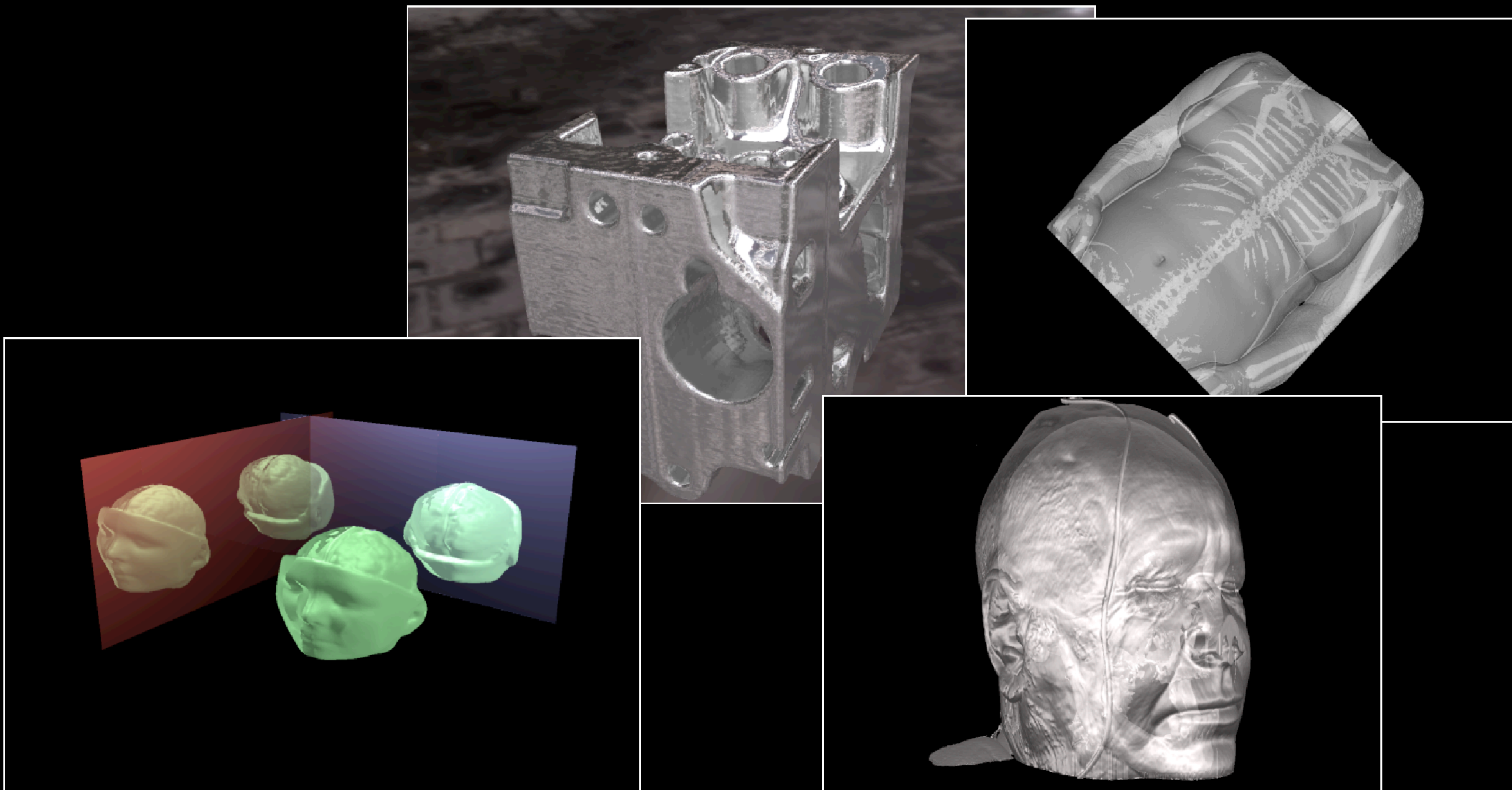
Reasons for Ray Tracing: Flexible Primitive Types

SIGGRAPH2005

Triangles, Bezier splines, and subdivision surfaces fully integrated

Reasons for Ray Tracing: Flexible Primitive Types

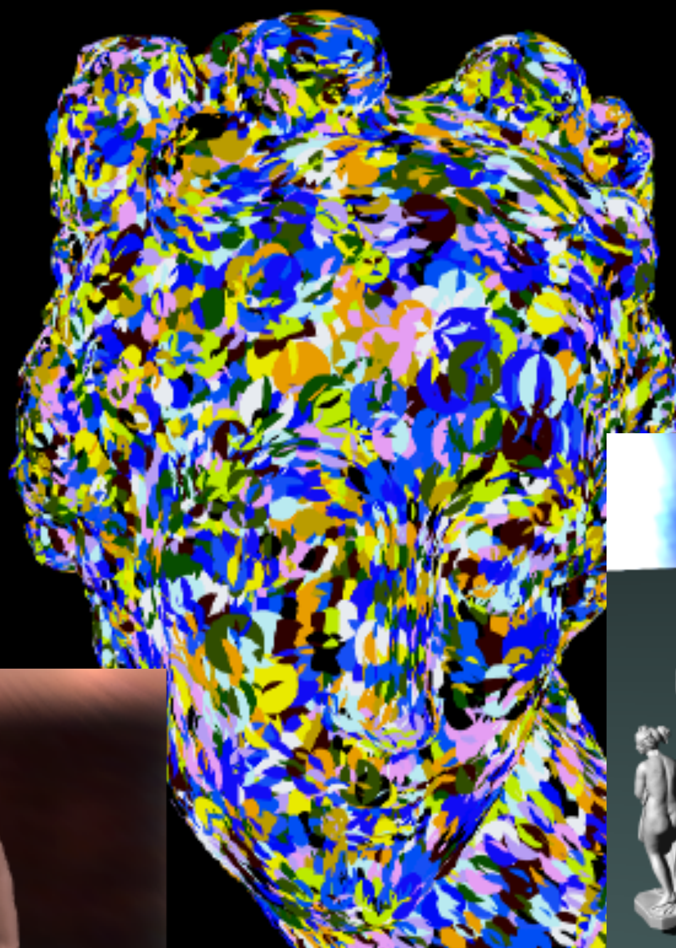
SIGGRAPH2005



Volume visualization using multiple iso-surfaces in combination with surface rendering

Reasons for Ray Tracing: Flexible Primitive Types

SIGGRAPH2005



24 MPoints, 2.1 fps with shadow @ 640x480

Realtime ray tracing of point clouds (1 Mpoints each)
On *one* dual-Opteron 2.4 GHz: 4-9 fps

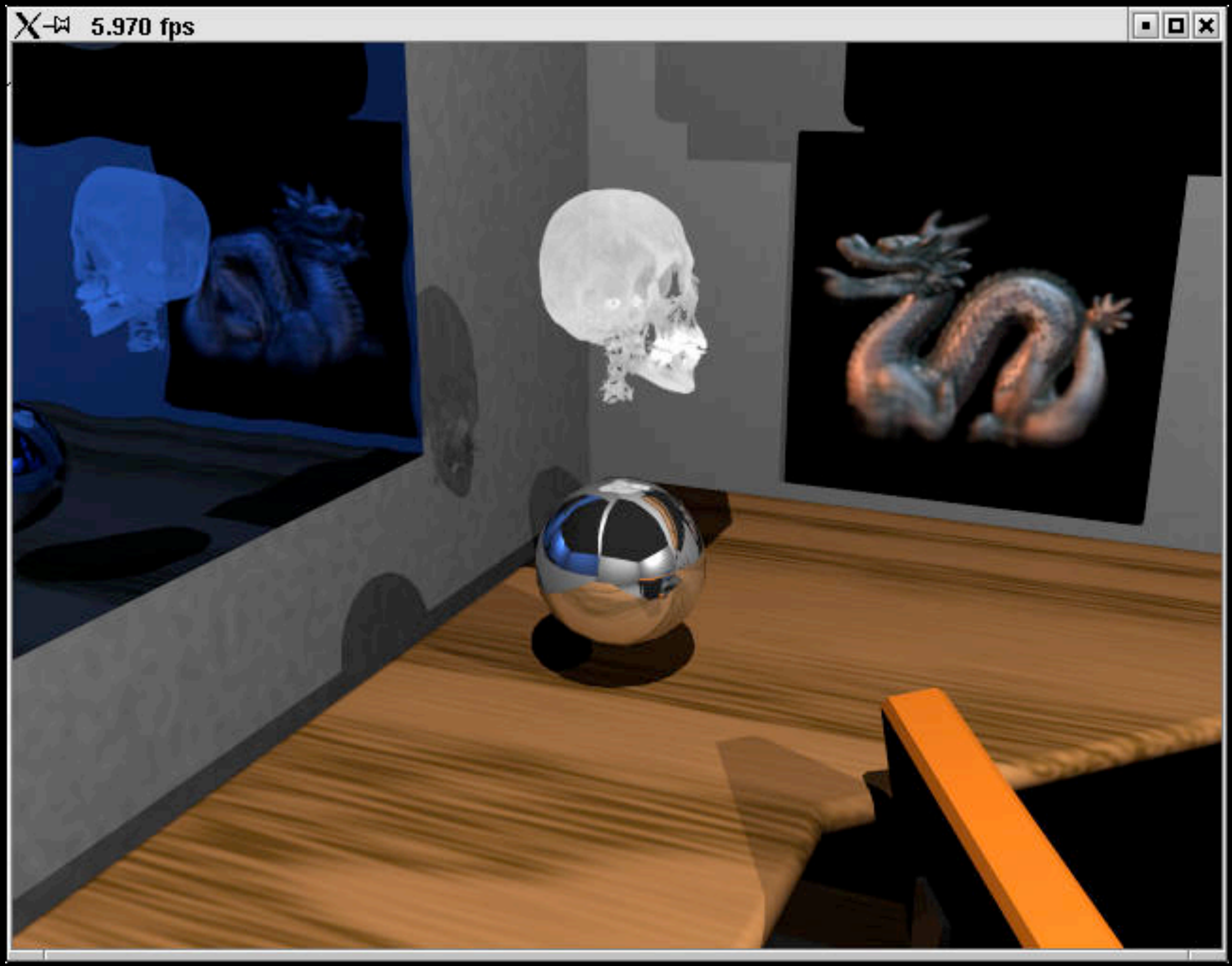
Reasons for Ray Tracing: Declarative Graphics

SIGGRAPH2005

- Declarative Graphics Interface
  - Application specifies scene once, plus updates
  - Rendering fully performed by renderer (e.g. in HW)
  - Similar to scene graphs, PostScript, or latest GUIs
- Benefits
  - Greatly simplifies application programming
  - Allows for complete HW acceleration

Reasons for Ray Tracing: Declarative Graphics

SIGGRAPH2005



Reasons for Ray Tracing: Declarative Graphics

SIGGRAPH2005

Reasons for Ray Tracing: Global Illumination

SIGGRAPH2005

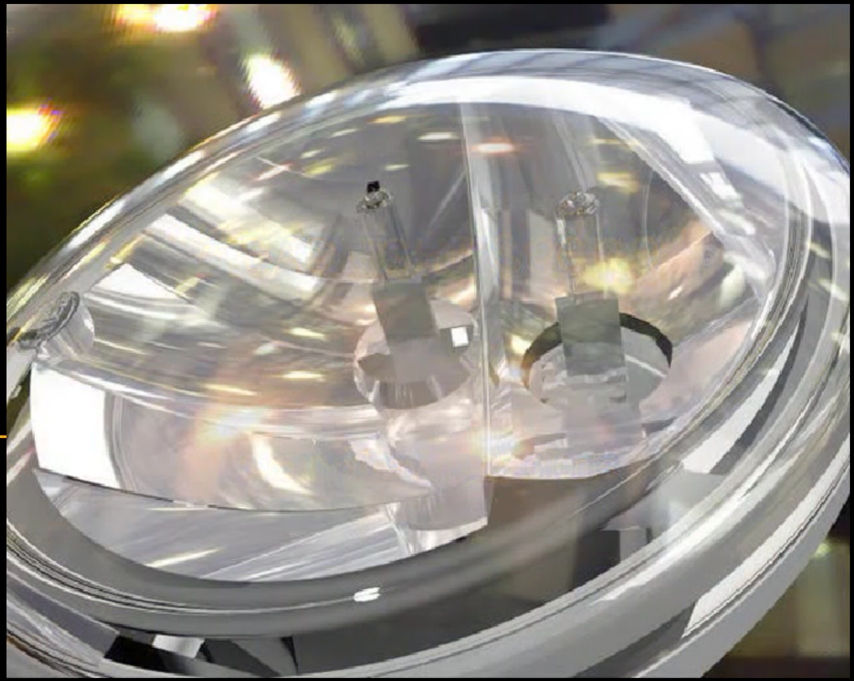
- Global Illumination
  - Simulating global lighting through tracing rays
  - Indirect diffuse and caustic illumination
  - Fully recomputed at up to 20 fps
- Benefits
  - Add the subtle but highly important clue for realism
  - Allows flexible light planning and control

Reasons for Ray Tracing: Global Illumination

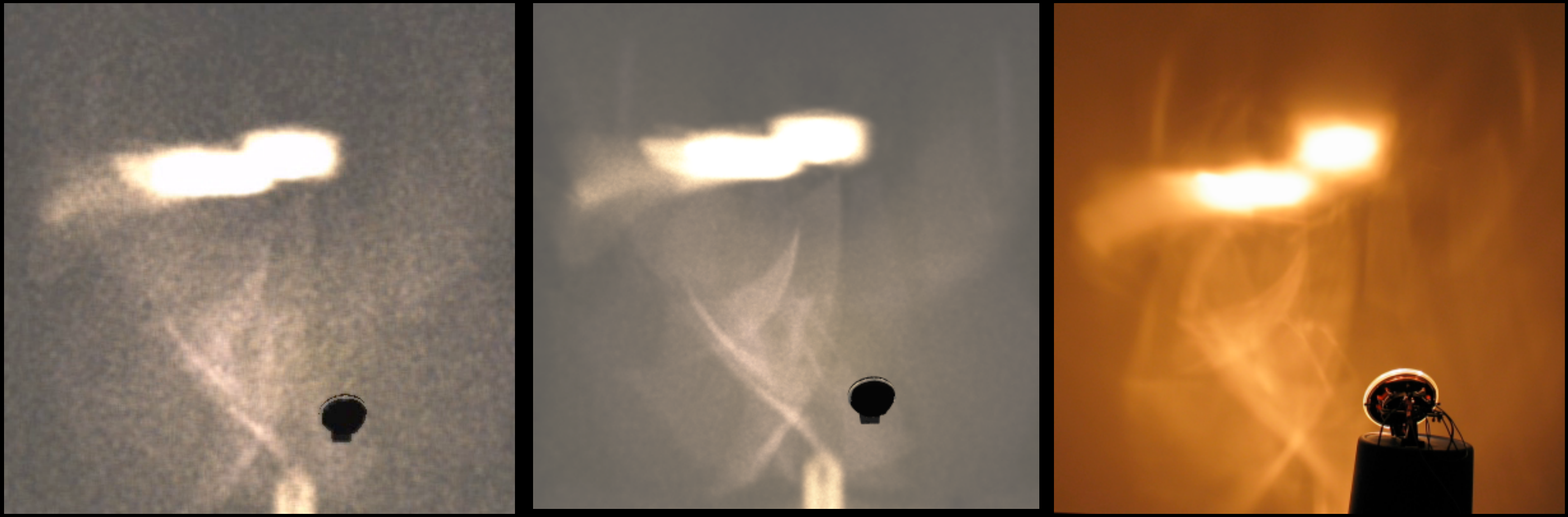
SIGGRAPH2005

Conference room (380 000 tris, 104 lights) with full global illumination in realtime

Reasons for Ray Tracing: Global Illumination



250k / 3 fps   25M / 11 fps   Photograph



Light pattern from a car head lamp computed in realtime using photon mapping:
Left: realtime update, middle: accumulated in 30s, right: photograph of real pattern

3D Vectors

Computer Graphics

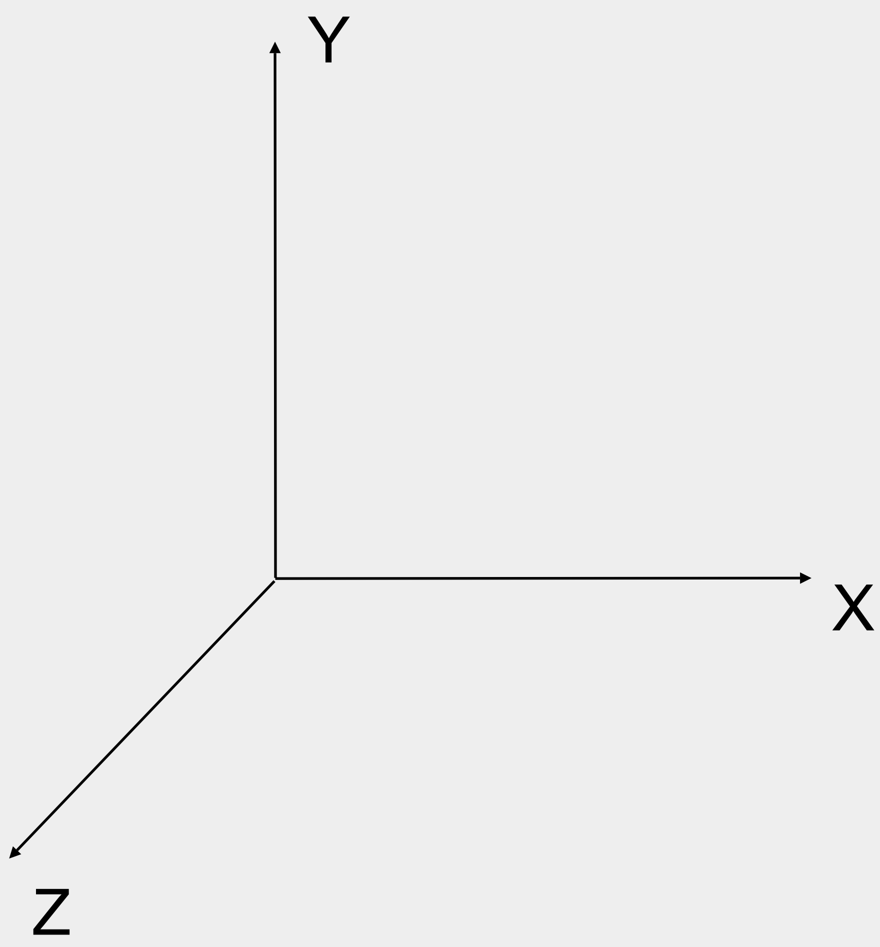
CSE167: Computer Graphics

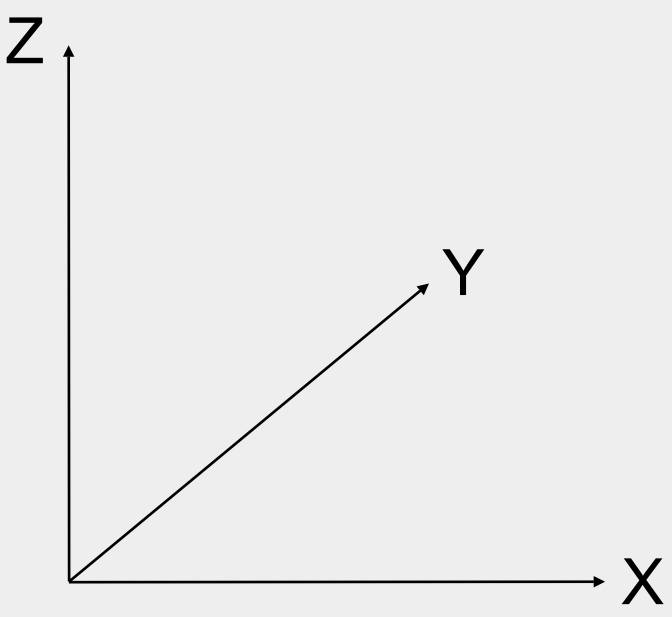
Instructor: Ronen Barzel

UCSD, Winter 2006

Coordinate Systems

- Right handed coordinate systems





- (more on coordinate systems next class)

Vector Arithmetic

$$\mathbf{a} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{bmatrix} \qquad \mathbf{a} - \mathbf{b} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix}$$

$$-\mathbf{a} = \begin{bmatrix} -a_x \\ -a_y \\ -a_z \end{bmatrix} \qquad s\mathbf{a} = \begin{bmatrix} sa_x \\ sa_y \\ sa_z \end{bmatrix}$$

Vector Magnitude

- The magnitude (length) of a vector is:

$$|\mathbf{v}|^2 = v_x^2 + v_y^2 + v_z^2$$

$$|\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

- A vector with length=1.0 is called a *unit vector*
- We can also *normalize* a vector to make it a unit vector:

$$\frac{\mathbf{v}}{|\mathbf{v}|}$$

Dot Product

$$\mathbf{a} \cdot \mathbf{b} = \sum a_i b_i$$

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

$$\mathbf{a} \cdot \mathbf{b} = |a||b| \cos\theta$$

Dot Product

$$\mathbf{a} \cdot \mathbf{b} = \sum a_i b_i$$

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

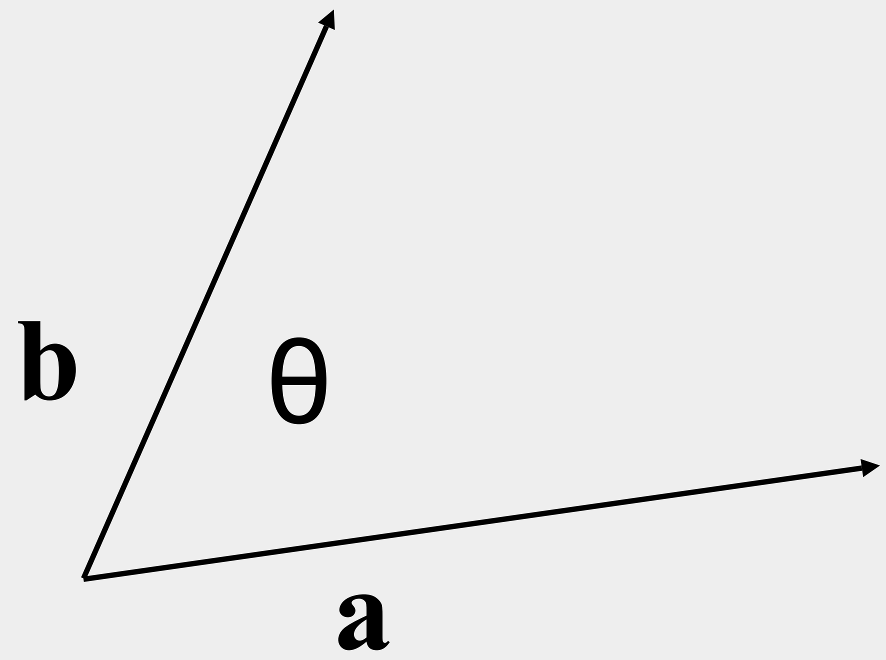
$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^{\mathrm{T}} \mathbf{b}$$

$$\mathbf{a} \cdot \mathbf{b} = |a||b| \cos \theta$$

Example: Angle Between Vectors

- How do you find the angle θ between vectors **a** and **b**?

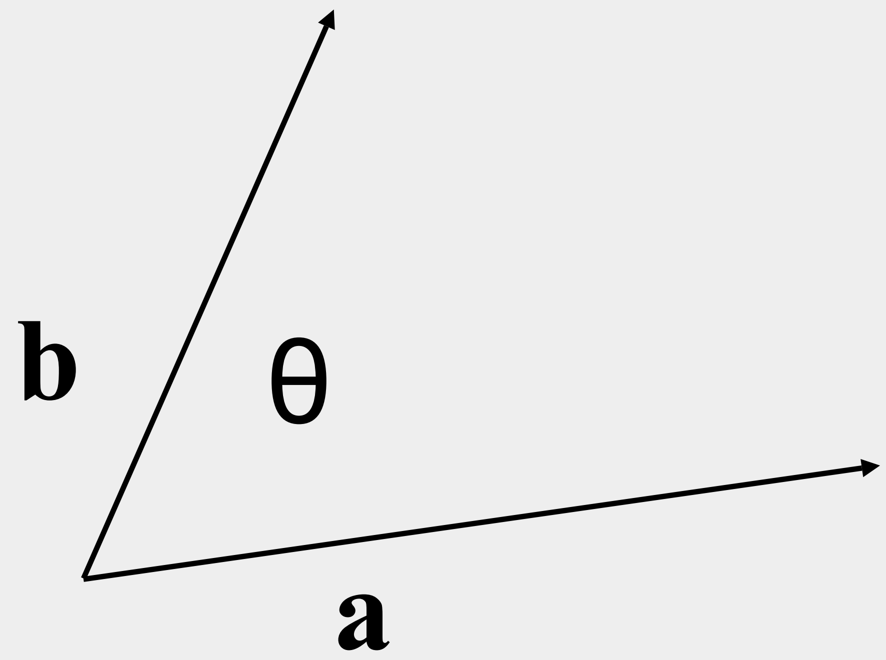


Example: Angle Between Vectors

$$\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

$$\cos\theta = \left(\frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right)$$

$$\theta = \cos^{-1}\left(\frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right)$$

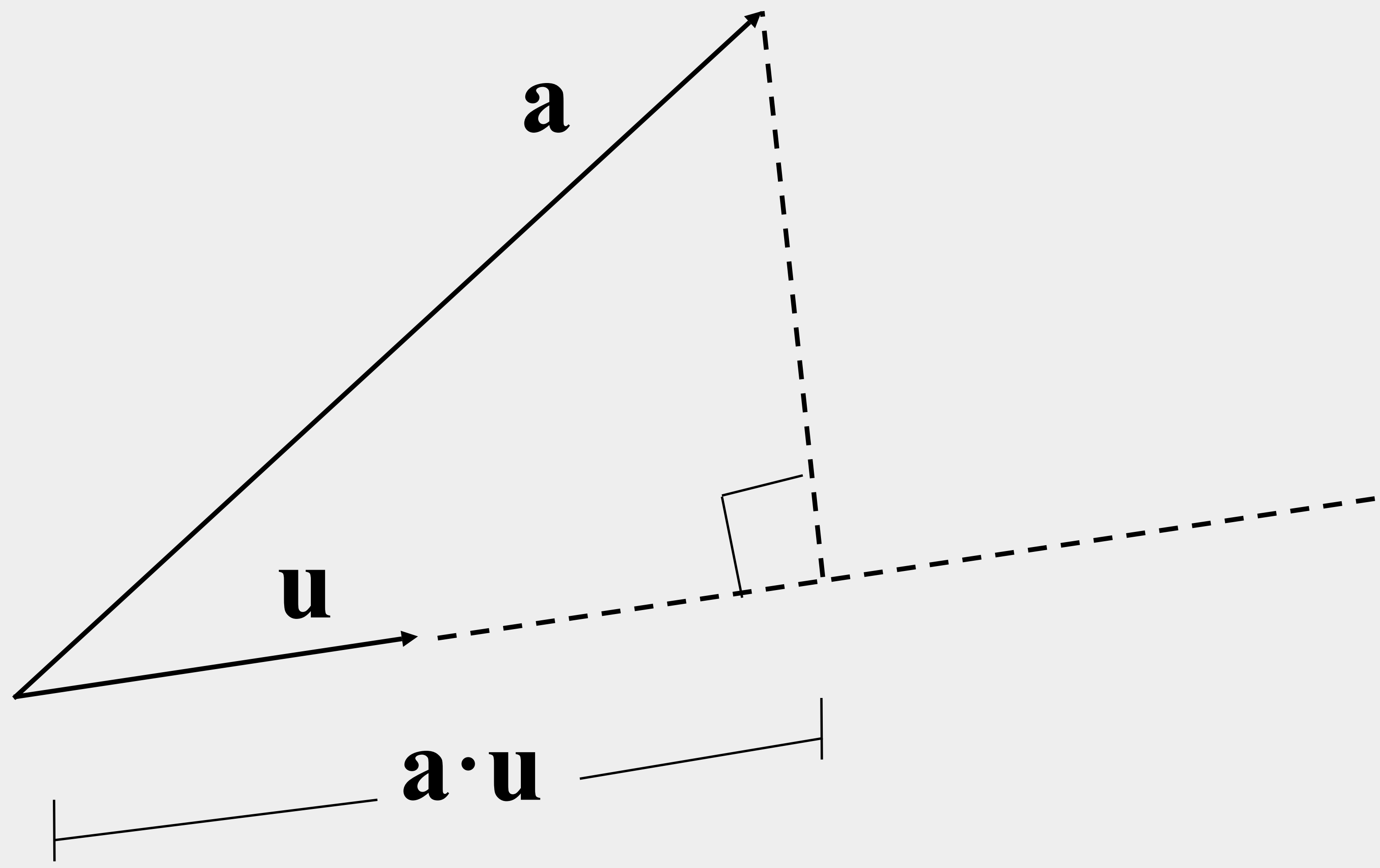


Dot Product Properties

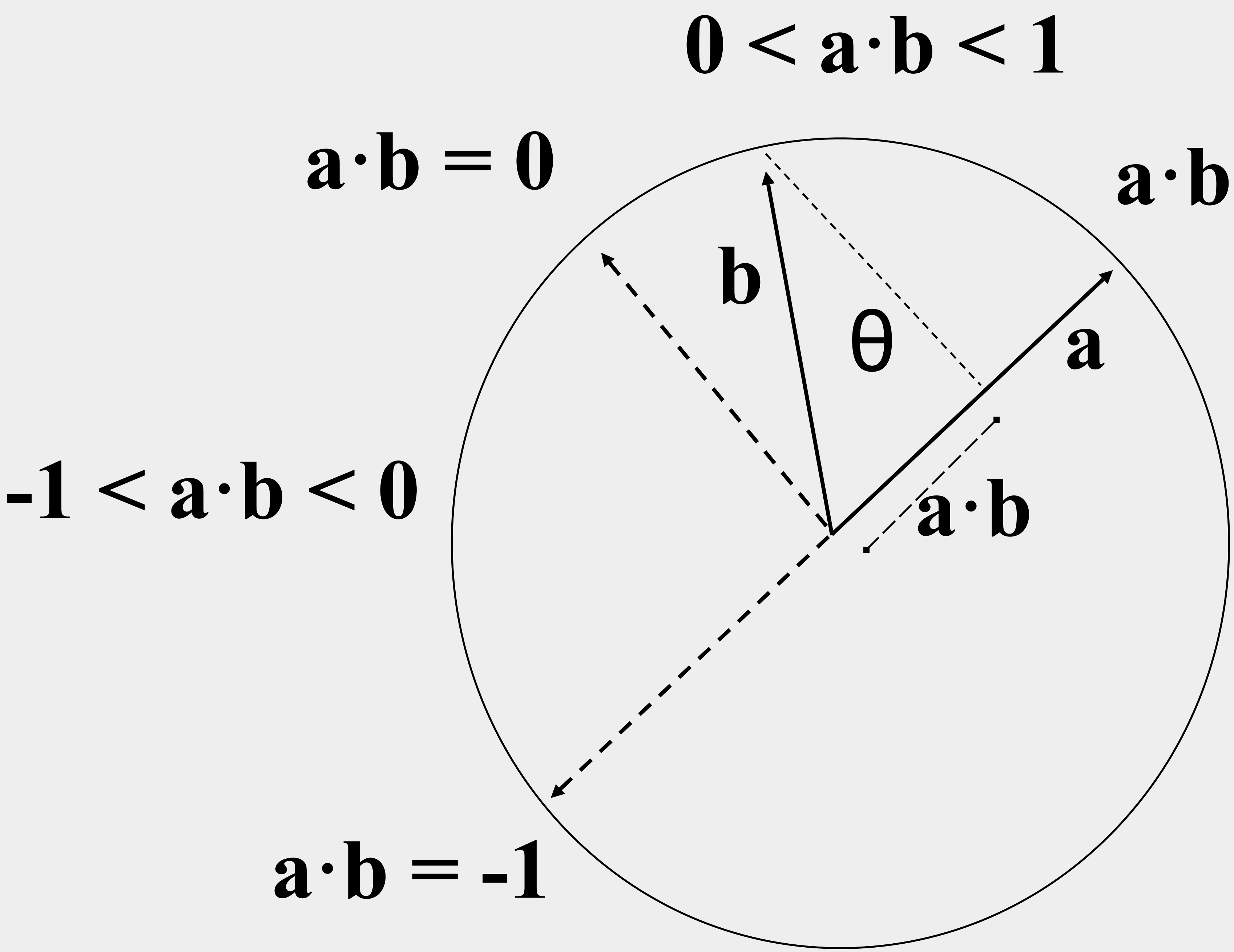
- The dot product is a scalar value that tells us something about the relationship between two vectors
  - If $\mathbf{a} \cdot \mathbf{b} > 0$ then $\theta < 90^\circ$
    - Vectors point in the same general direction
  - If $\mathbf{a} \cdot \mathbf{b} < 0$ then $\theta > 90^\circ$
    - Vectors point in opposite direction
  - If $\mathbf{a} \cdot \mathbf{b} = 0$ then $\theta = 90^\circ$
    - Vectors are perpendicular
    - (or one or both of the vectors is degenerate (0,0,0))

Dot Products with One Unit Vector

- If $|\mathbf{u}|=1.0$ then $\mathbf{a}\cdot\mathbf{u}$ is the length of the *projection* of **a** onto **u**



Dot Products with Unit Vectors



$$|\mathbf{a}| = |\mathbf{b}| = 1.0$$

$$\mathbf{a} \cdot \mathbf{b} = \cos(\theta)$$

Cross Product

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} a_y b_z - a_z b_y & a_z b_x - a_x b_z & a_x b_y - a_y b_x \end{bmatrix}$$

Properties of the Cross Product

$\mathbf{a} \times \mathbf{b}$ is a *vector* perpendicular to both **a** and **b**, in the direction defined by the right hand rule

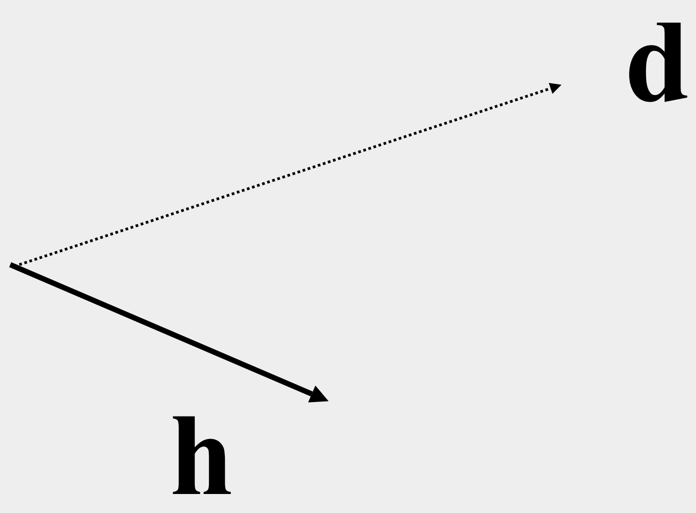
$$\left|\mathbf{a} \times \mathbf{b}\right| = \left|\mathbf{a}\right|\left|\mathbf{b}\right|\sin\theta$$

$\left|\mathbf{a} \times \mathbf{b}\right| =$ area of parallelogram **ab**

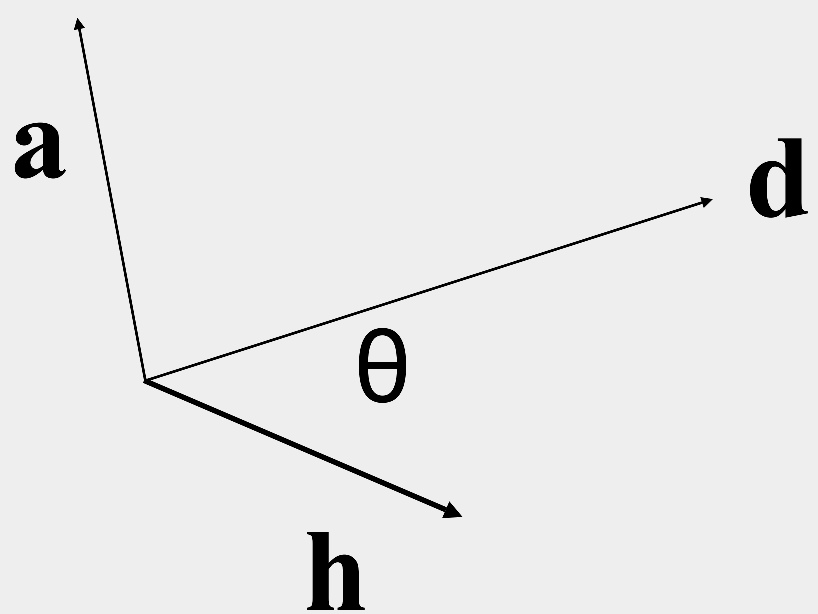
$\left|\mathbf{a} \times \mathbf{b}\right| = 0$ if **a** and **b** are parallel (or one or both degenerate)

Example: Align two vectors

- We are heading in direction **h**. We want to rotate so that we will align with a different direction **d**. Find a unit axis **a** and an angle θ to rotate around.



Example: Align two vectors



$$\mathbf{a} = \frac{\mathbf{h} \times \mathbf{d}}{\left|\mathbf{h} \times \mathbf{d}\right|}$$

$$\theta = \sin^{-1}\left(\frac{\left|\mathbf{h} \times \mathbf{d}\right|}{\left|\mathbf{h}\right|\left|\mathbf{d}\right|}\right)$$

$$\theta = \cos^{-1}\left(\frac{\mathbf{h} \cdot \mathbf{d}}{\left|\mathbf{h}\right|\left|\mathbf{d}\right|}\right)$$

$$\theta = \tan^{-1}\left(\frac{\left|\mathbf{h} \times \mathbf{d}\right|}{\mathbf{h} \cdot \mathbf{d}}\right)$$

$$\texttt{theta} = \texttt{atan2}\left(\left|\mathbf{h} \times \mathbf{d}\right|, \mathbf{h} \cdot \mathbf{d}\right)$$

Float3 Structure

```
struct Float3
{
    union
    {
        float d[3];
        struct { float x, y, z; };
        struct { float r, g, b; };
        struct { float s, t, u; };
        struct { float alpha, beta, gamma; };
    };

    :
    :
    :

}
```

Float3 Structure

```
struct Float3
{
    :
    :
    Float3();
    Float3(float c);
    Float3(int i);
    Float3(double x);
    Float3(float x, float y, float z);

    const float &operator[](int i) const
        { return d[i]; }
    float &operator[](int i)
        { return d[i]; }
    :
    :
}
```

Float3 Structure

```
struct Float3
{
    :
    :
    Float3 &operator += (const Float3 &x)
        { FOR(i,3) d[i] += x[i];  return *this; }
    Float3 &operator -= (const Float3 &x)
        { FOR(i,3) d[i] -= x[i];  return *this; }
    Float3 &operator *= (const Float3 &x)
        { FOR(i,3) d[i] *= x[i];  return *this; }
    Float3 &operator *= (const float &x)
        { FOR(i,3) d[i] *= x;     return *this; }
    Float3 &operator /= (const Float3 &x)
        { FOR(i,3) d[i] /= x[i];  return *this; }
    Float3 &operator /= (const float &x)
        { FOR(i,3) d[i] /= x;     return *this; }
    :
    :
}
```

Float3 Structure

```
struct Float3
{
    :
    :
    Float3 operator + (const Float3 &x) const
        { return Float3(d[0] + x[0], d[1] + x[1], d[2] + x[2]); }
    Float3 operator - () const
        { return Float3(-d[0], -d[1], -d[2]); }
    Float3 operator - (const Float3 &x) const
        { return Float3(d[0] - x[0], d[1] - x[1], d[2] - x[2]); }
    Float3 operator * (const Float3 &x) const
        { return Float3(d[0] * x[0], d[1] * x[1], d[2] * x[2]); }
    Float3 operator * (float x) const
        { return Float3(d[0]*x, d[1]*x, d[2]*x); }
    Float3 operator / (const Float3 &x) const
        { return Float3(d[0] / x[0], d[1] / x[1], d[2] / x[2]); }
    Float3 operator / (float x) const {
        float inv = 1.0f / x;
        return Float3(d[0] * inv, d[1] * inv, d[2] * inv);
    }
    :
    :
}
```

Float3 Structure

```
struct Float3
{
    :
    :
    float length_squared() const
    {
        return d[0]*d[0] + d[1]*d[1] + d[2]*d[2];
    }

    float length() const
    {
        return sqrtf(length_squared());
    }
    :
    :
}
```

Float3 Structure

```
inline Float3 normalize(const Float3 &v)
{
    return v / v.length();
}

inline Float3 operator*(float f, const Float3 &v)
{
    return v*f;
}

inline float dot(const Float3 &v1, const Float3 &v2)
{
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}

inline Float3 cross(const Float3 &v1, const Float3 &v2)
{
    return Float3((v1.y * v2.z) - (v1.z * v2.y),
                  (v1.z * v2.x) - (v1.x * v2.z),
                  (v1.x * v2.y) - (v1.y * v2.x));
}
```

Elementary Ray Tracer

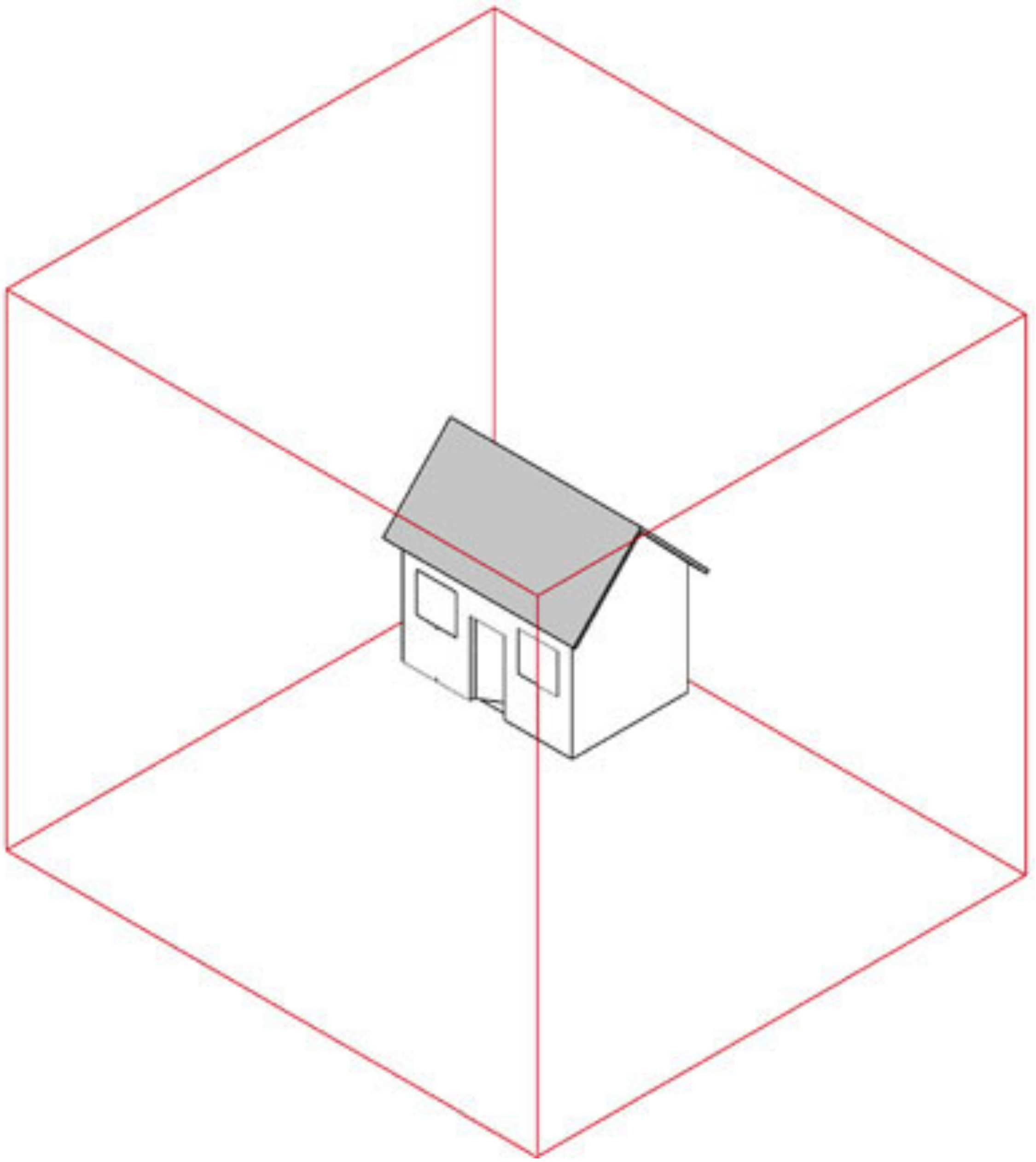
Projection

- Models are 3D, but images are 2D.
- The process of converting 3D to 2D is called **projection**.
- Typically, computer graphics apps use two kinds of projections.
    - Orthographic Projection
    - Perspective Projection
- We will use *orthographic projection* here.

Orthographic Projection

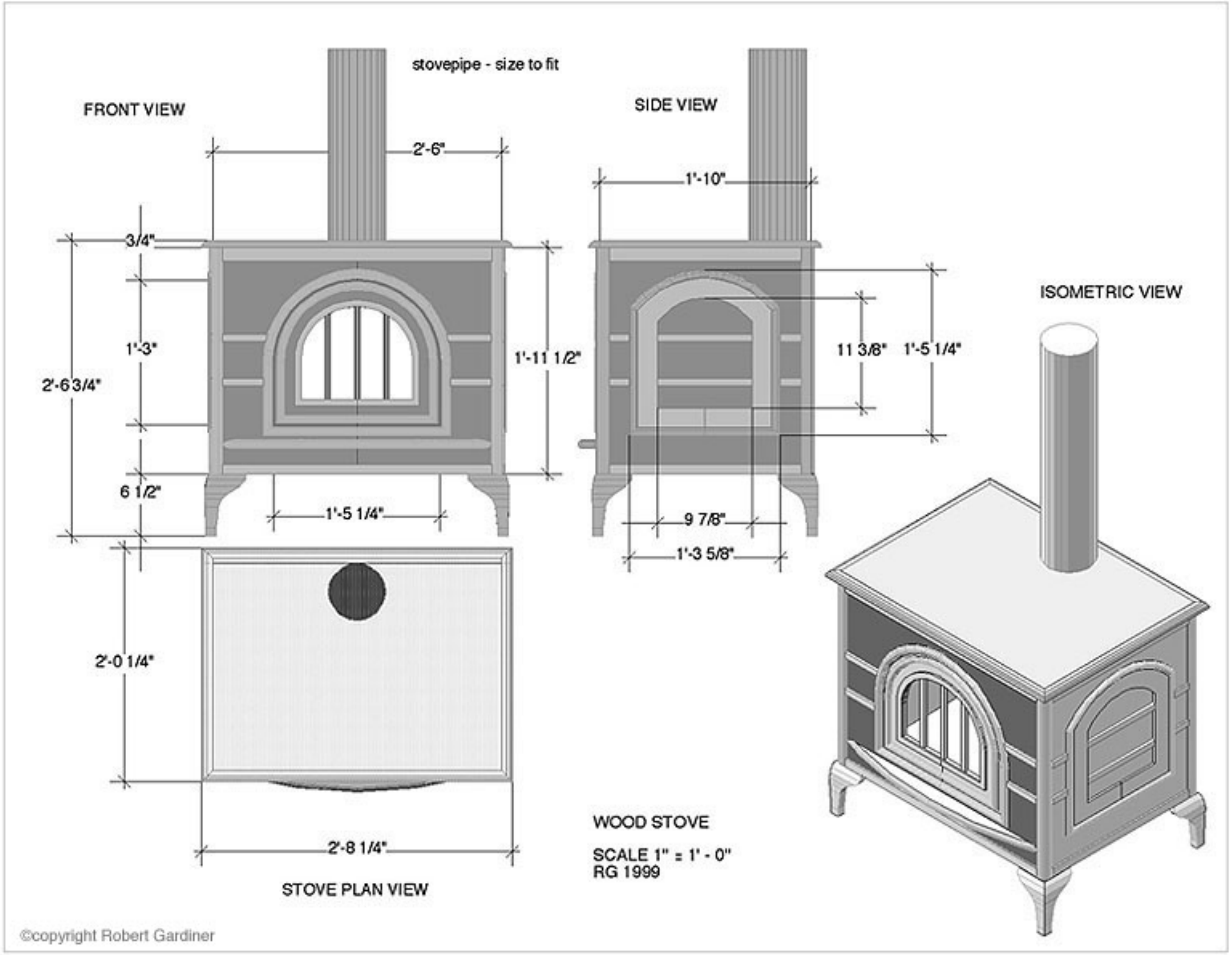
- Pixels are on an **image plane.**
- Rays are *perpendicular* to the plane.
- Lacks **foreshortening** --- further objects do not get smaller.
- Used in design/architectural drawings, where precision is important.
- Human eyes do not see this way.

Orthographic Projection



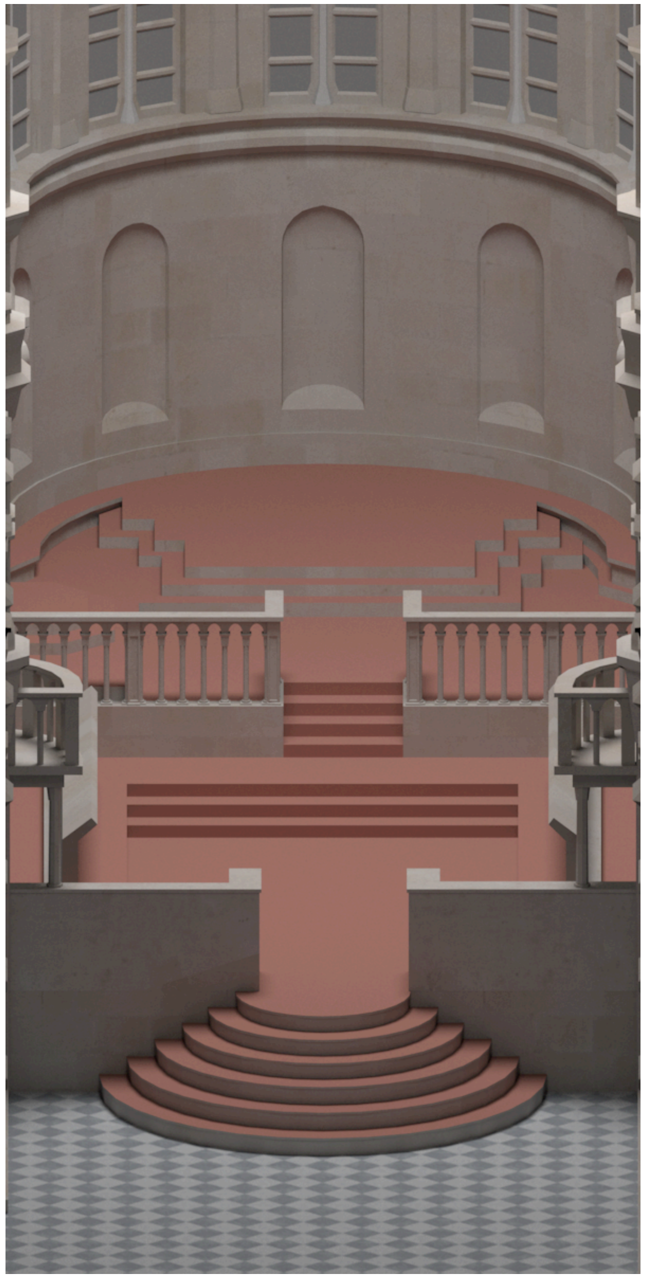
http://www2.arts.ubc.ca/TheatreDesign/crslib/drft_1/orthint.htm

Orthographic Projection

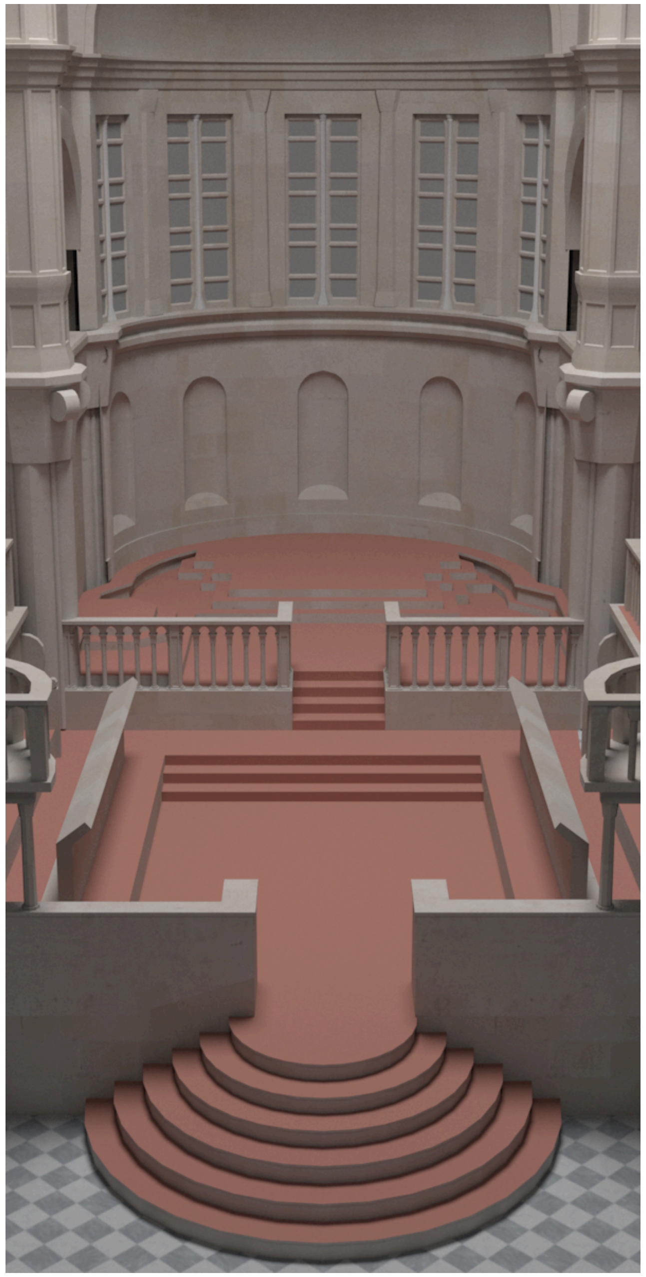


http://www2.arts.ubc.ca/TheatreDesign/crslib/drft_1/cad/wdstv.htm

Orthographic vs Perspective



orthographic

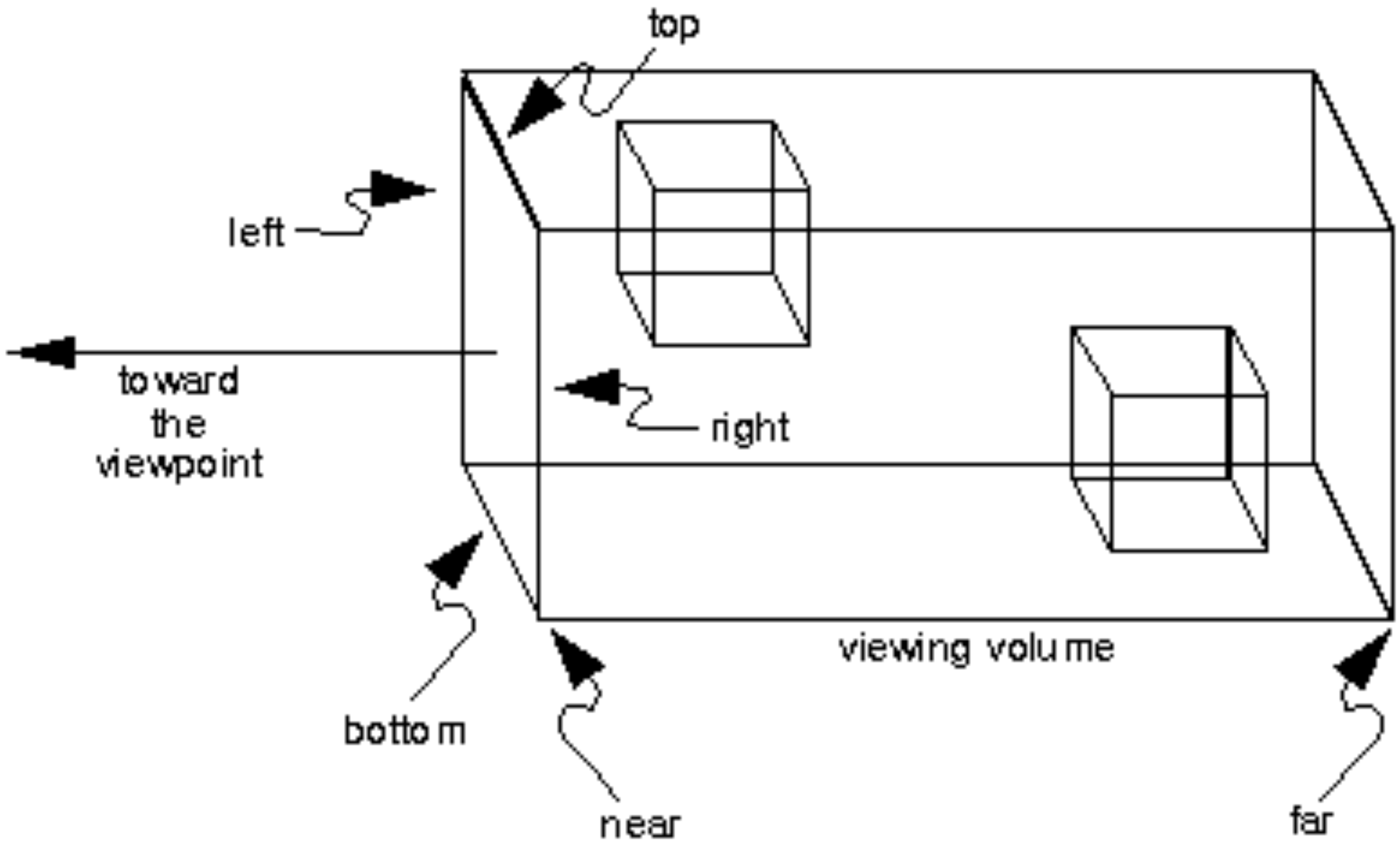


perspective

Defining Orthographic Projection

- We need six parameters
  - **left, right**: bounds in x-axis
  - **top, bottom**: bounds in y-axis
  - **hither (near), yon (far)**: bounds in z-axis
- This define a *prism* inside which are the things we see.

Orthographic Prism

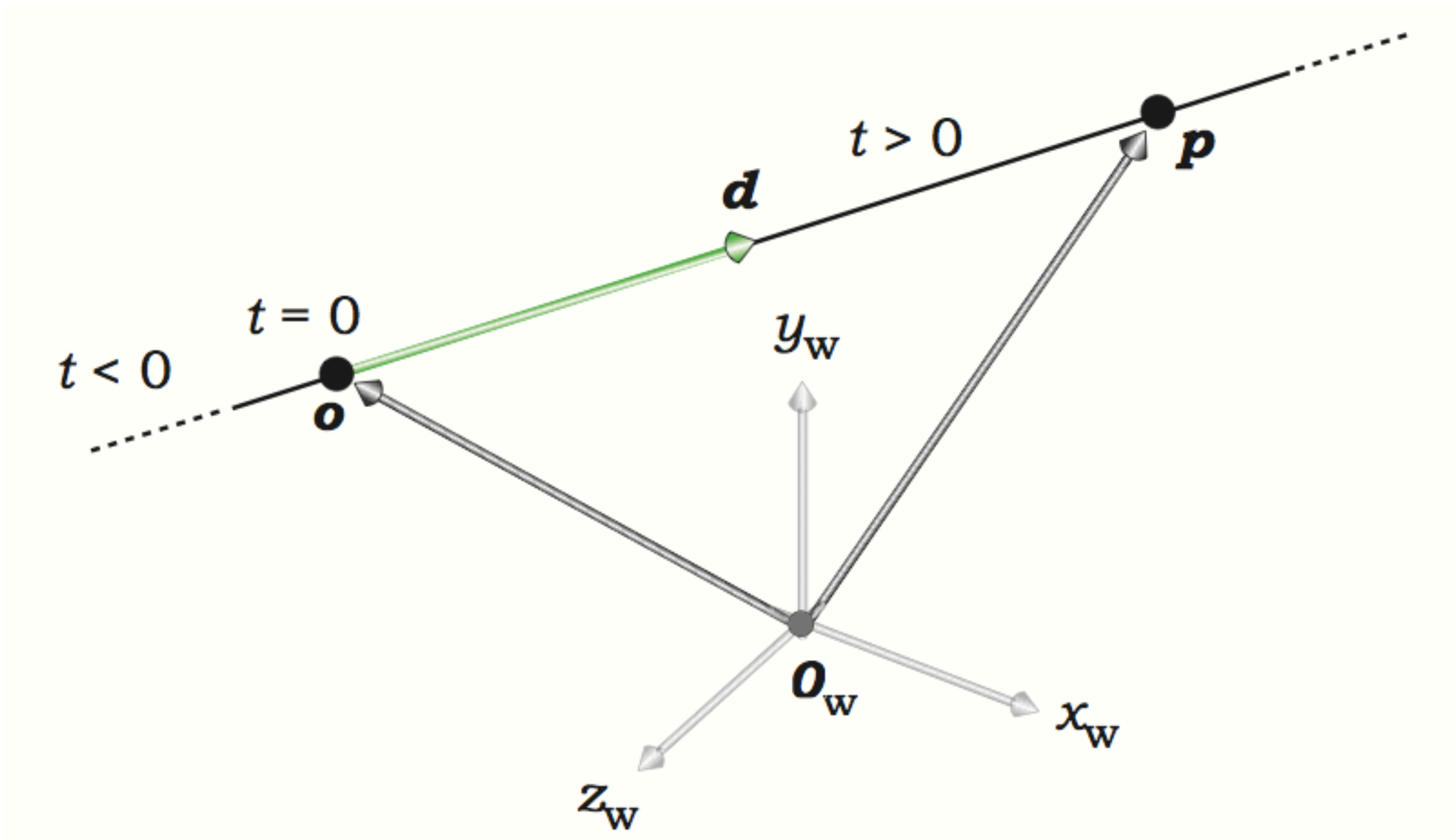


Ray

- Basically, it's a *half line*.
  - Begins at a point called the **origin**.
  - Extends towards infinity in a given **direction**.
    - For simplicity, direction should always be a *unit vector*.
- Let the origin be denoted by **o** and let the direction be denoted by **d**.
- Then, a ray is a set of the following points:

$$\{\mathbf{o} + t\mathbf{d} : t \in [0, \infty]\}$$

Ray



$\boldsymbol{d}$
$t > 0$
$\boldsymbol{p}$
$t = 0$
y_w
$t < 0$
$\boldsymbol{o}$
$\boldsymbol{o}_w$
x_w
z_w

Ray

```
struct Ray
{
    Float3 origin;
    Float3 direction;
    float tmin;
    float tmax;

    Ray(const Float3 &_origin = Float3(0,0,0),
        const Float3 &_direction = Float3(0,0,1),
        float _tmin = 0,
        float _tmax = INFINITY);
    ~Ray();

    inline Float3 operator() (float t) const
    {
        return origin + direction * t;
    }
};
```

Ray

- tmin
  - The time the ray starts
  - Typically 0 or a very small value, say 0.00001.
  - We use 0.00001 because we want to avoid the case where the view plane is on the surface of something

- tmax
  - The time the ray stops.
  - The time it hits something.
  - Initially, infinity.

- Set of points (revisited)

$$\{\mathbf{o} + t\mathbf{d} : t_{\min} \leq t < t_{\max}\}$$

Camera

- Responsible for generating rays.
- Let's define image plane to be the rectangle

$$\{(x,y) : -1 \leq x, y \leq 1\}$$

- A camera maps (x,y) from the image plane to a ray.
- ```
  class Camera
  {
  public:
      Camera();
      virtual ~Camera();

      virtual Ray gen_ray(float sx, float sy) const = 0;
  };
  ```
  ```

Orthographic Camera

- Needs 6 parameters: left, right, bottom, top, hither, yon

- ```
  class OrthographicCamera : public Camera
  {
  public:
      OrthographicCamera(
          float _left,
          float _right,
          float _bottom,
          float _top,
          float _hither = RAY_EPSILON,
          float _yon = INFINITY);
      virtual ~OrthographicCamera();
      virtual Ray gen_ray(float sx, float sy) const;

  public:
      float left, right, bottom, top, hither, yon;
  };
  ```
  ```

Orthographic Camera

- Image plane is the *xy-plane*.
- So the ray must travel along the *z-axis*.
- Orthographic camera generates rays that travels in the *negative-z* direction
- ```
  Ray OrthographicCamera::gen_ray( float sx, float sy ) const
  {
      sx = 0.5f + sx / 2;
      sy = 0.5f + sy / 2;

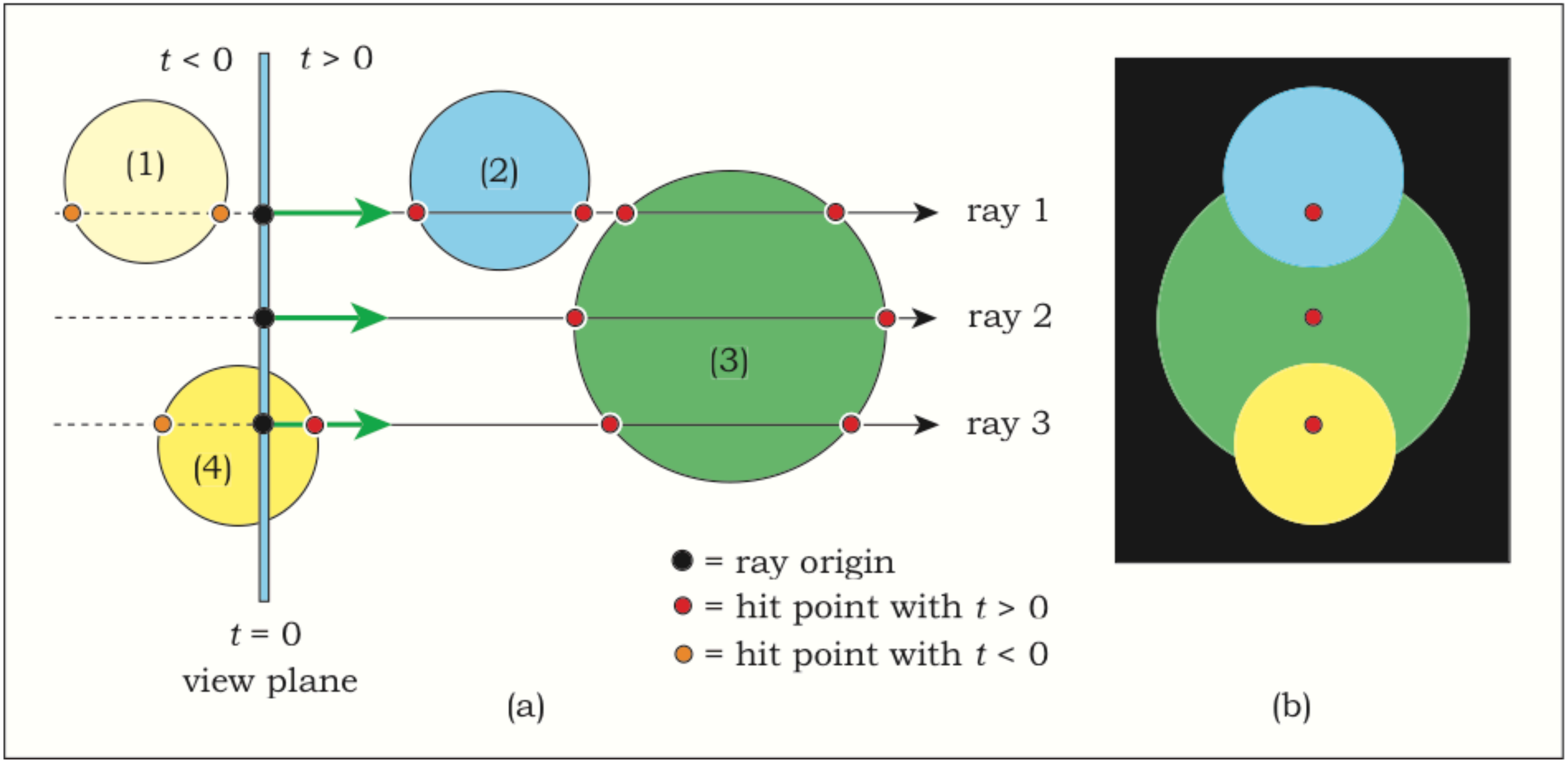
      float x = left + (right - left) * sx;
      float y = bottom + (top - bottom) * sy;

      return Ray(Float3(x,y,hither), Float3(0,0,-1), 0.00001f, yon-hither);
  }
  ```
  ```

Ray-Object Intersection

- In a scene, there are several objects.
- For each ray, we have to find the *nearest object* that the ray hits.
- But the hit time t must be greater than 0.

Ray-Object Intersection



Geometric Shapes

- They are models of objects in the scene.
- Each object must be able to
  - Tell if a ray intersects it
  - Compute the time of intersection
- In this lecture, each object has one color.
  - We’re going work with two more sophisticated models later.

Class Shape

- ```cpp
  class Shape
  {
  public:
      Shape(const Float3 &_color);
      virtual ~Shape();
      virtual bool intersect_p(Ray &ray) = 0;

      Float3 color;
  };
  ```

- intersect_p
  - Return true if the given ray intersects the shape.
  - Modify the ray's tmax to the intersection if intersection occurs.
  ```

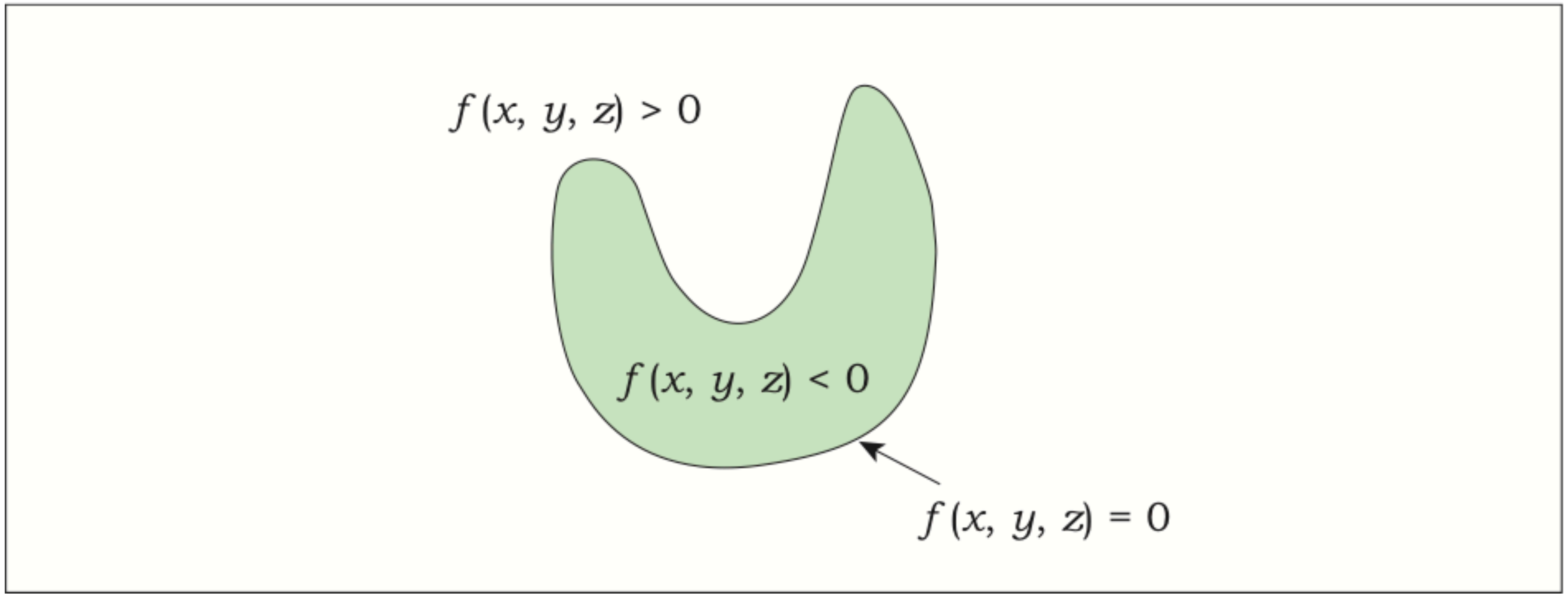
Definitions of Sets

- A set can be defined in two ways.
  - Explicitly: As the set of images of a functions of free variables.
    - A line can be defined as $\{\mathbf{o} + t\mathbf{d} : t \in \mathbb{R}\}$
    - A unit circle can be defined as $\{(\cos\theta, \sin\theta) : \theta \in [0, 2\pi)\}$
  - Implicitly: As the set that satisfies a certain conditions.
    - A line can be defined as $\{(x, y) : Ax + By + C = 0\}$
    - A unit circle can be defined as $\{(x, y) : x^2 + y^2 = 1\}$

Implicit Definition

- Typically, implicit definitions has a function f that takes in a point and produces a real number.

- Implicit surface is defined as *all points at which the function evaluates to 0*.
  - If the value is greater than 0, the point is said to be outside.
  - If the value is less than 0, the point is said to be inside.



Definition of Sets in Computer Graphics

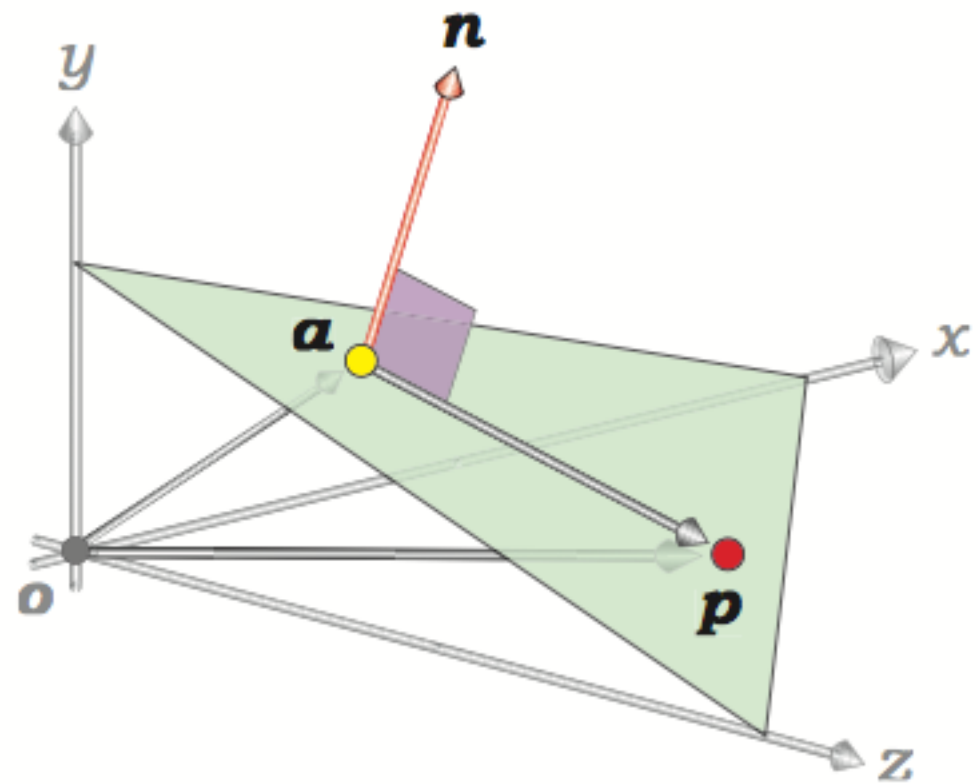
- We define rays *explicitly*.
- We define shapes *implicitly.*
- Why? Because it helps with ray-shape intersection.
- Say, we have a shape defined as $\{\mathbf{p} : f(\mathbf{p}) = 0\}$
  And we want to intersect it with ray $\{\mathbf{o} + t\mathbf{d} : t_{\min} \leq t < t_{\max}\}$
- We just have to solve the following equation for t:

$$f(\mathbf{o} + t\mathbf{d}) = 0$$

- Then we can decide whether t is in range or not.

Plane

- Infinite flat sheet of points.
- Defined by
  - A point **a**
  - Normal vector **n**



- Plane is the set of points **p** such that the vector from **a** to **p** is *perpendicular* to the normal.

$$\{\mathbf{p} : (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0\}$$

Plane Class

- ```
  class Plane : public Shape
  {
  public:
      Plane(const Float3 &_point, const Float3 &_normal, const Float3 &_color);
      virtual ~Plane();
      virtual bool intersect_p(Ray &ray);

  public:
      Float3 point;
      Float3 normal;
  };
  ```

- Here, the field “point” is the point **a** on the plane.
  And the field “normal” is the normal vector **n**.
  ```

Ray-Plane Intersection

- We substitute $\mathbf{p}$ with $\mathbf{o} + t\mathbf{d}$ in the plane's equation:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0$$

- Solving for t, we have

$$t = \frac{(\mathbf{a} - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

Ray-Plane Intersection

```
bool Plane::intersect_p( Ray &ray )
{
    float A = dot(ray.origin, normal);
    float B = dot(ray.direction, normal);
    float C = dot(point, normal);
    float t = (C - A) / B;

    if (t >= ray.tmin && t < ray.tmax)
    {
        ray.tmax = t;
        return true;
    }
    else
        return false;
}
```

Sphere

- A set of points that are of a constant from a point called the **center** (**c**).
  - The constant distance is called the **radius** (r).
- Set of points:

$$\{\mathbf{p} : \|\mathbf{p} - \mathbf{c}\| = r\}$$

$$\{\mathbf{p} : (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) = r^2\}$$

- However, if we say that $\mathbf{c} = (c_x, c_y, c_z)$ then the above definition becomes:

$$\{(x, y, z) : (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2\}$$

Sphere Class

```
class Sphere : public Shape
{
public:
    Sphere(const Float3 &_center, float _radius, const Float3 &_color);
    virtual ~Sphere();
    virtual bool intersect_p(Ray &ray);

public:
    Float3 center;
    float radius;
};
```

Ray-Sphere Intersection

- Substituting $\mathbf{o} + t\mathbf{d}$ into the second set definition yields:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - r^2 = 0$$

- Expanding, we have

$$(\mathbf{d} \cdot \mathbf{d})t^2 + [2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}]t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0$$

- The above equation is a quadradic equation $at^2 + bt + c = 0$ where
  - $a = \mathbf{d} \cdot \mathbf{d}$
  - $b = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}$
  - $c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$

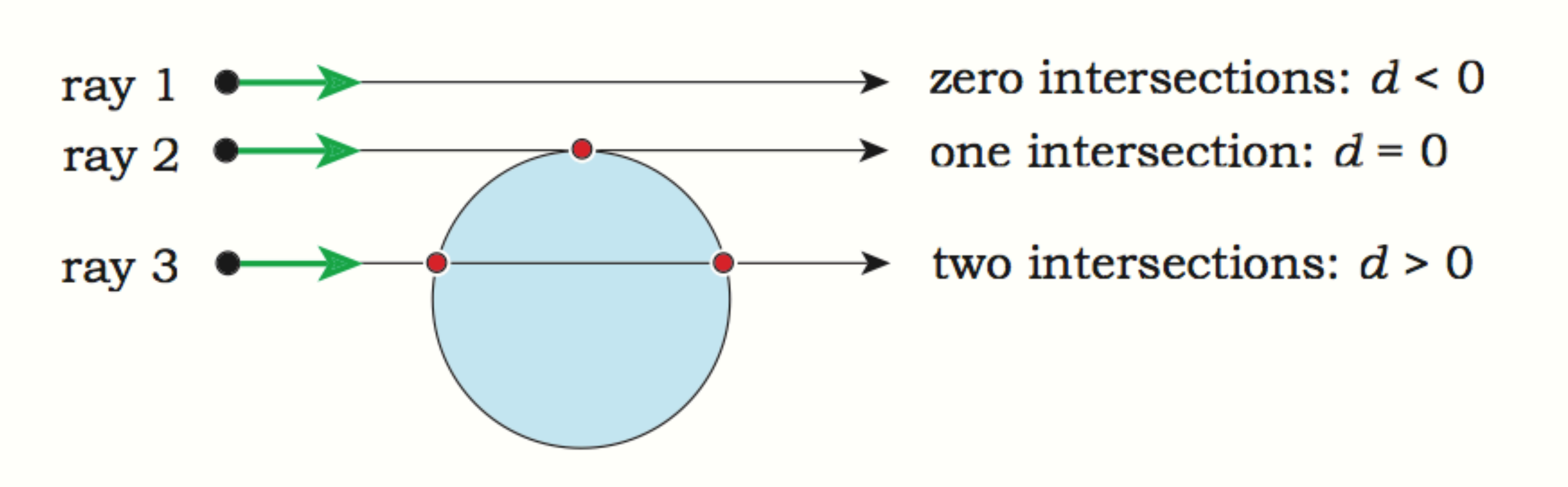
Ray-Sphere Intersection

- We can solve the quadratic equation and get

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

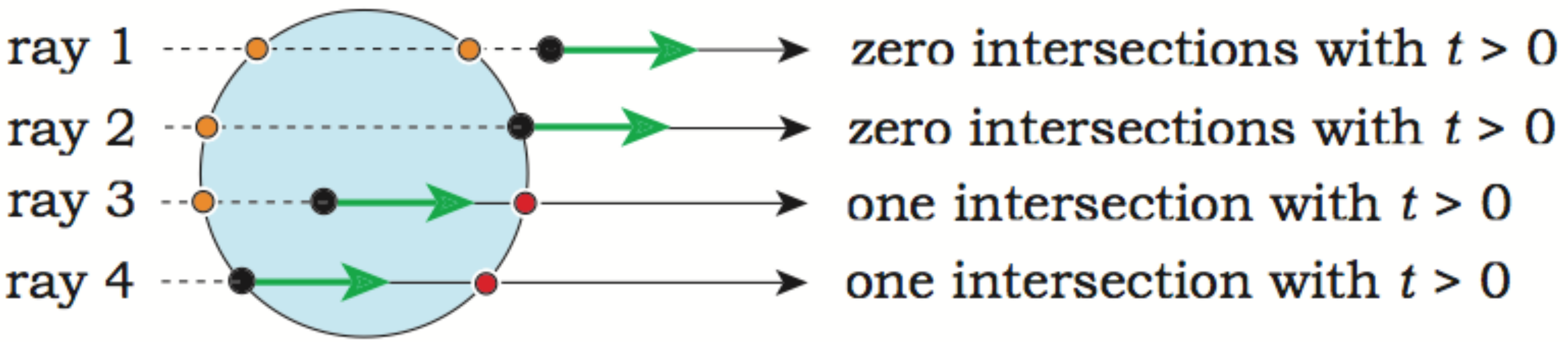
- The **discriminant** $d = b^2 - 4ac$ tells us how the ray intersects the sphere.
  - If d < 0, the ray doesn't intersect the sphere.
  - If d = 0, the ray intersects the sphere at only one point.
  - If d > 0, the ray intersects the sphere at two points.

Ray-Sphere Intersection



Ray-Sphere Intersection

- After computing t, we're not done. We have to find the *least non-negative t*.



Ray-Sphere Intersection

```
bool Sphere::intersect_p( Ray &ray )
{
    float t;
    Float3 temp = ray.origin - center;
    float a = dot(ray.direction, ray.direction);
    float b = 2 * dot(temp, ray.direction);
    float c = dot(temp, temp) - radius * radius;
    float disc = b*b - 4*a*c;

    if (disc < 0.0f)
        return false;

          :
          :
          :
```

Ray-Sphere Intersection

```
    else
    {
        float e = sqrtf(disc);
        float denom = 2.0f * a;
        t = (-b - e) / denom;

        if (t >= ray.tmin && t < ray.tmax)
        {
            ray.tmax = t;
            return true;
        }

        t = (-b + e) / denom;
        if (t >= ray.tmin && t < ray.tmax)
        {
            ray.tmax = t;
            return true;
        }
        else
            return false;
    }
}
```

Scene

- A scene is a combination of two things.
  - A number of shapes.
  - The camera.
- ```
  class Scene
  {
  public:
      Scene(Camera *_camera = NULL);
      virtual ~Scene();

  public:
      Camera *camera;
      std::vector<Shape *> shapes;
  };
  ```
  ```

Rendering a Scene

- Pseudocode:

```
For row = 0 to image_width-1 do
  For col = 0 to image_height-1 do

    1. Convert (row,col) to (x,y) where -1 <= x,y <= 1

    2. Use camera to generate ray from (x,y)

    3. Find the first object the ray intersects

    4. Record the color of the object to the image
```

Rendering a Scene

```
FOR(iy, image_height)
FOR(ix, image_width)
{
    float sx = 2 * (ix + 0.5f) / image_width - 1;
    float sy = 2 * (iy + 0.5f) / image_height - 1;

    Ray ray = scene.camera->gen_ray(sx, sy);

    Shape *hitted_shape = NULL;
    FOR(shape_index, shape_count)
    {
        Shape *shape = scene.shapes[shape_index];
        if (shape->intersect_p(ray))
            hitted_shape = shape;
    }

    if (hitted_shape != NULL)
        image[ix, iy] = hitted_shape->color;
    else
        image[ix, iy] = background_color;
}
```