Geometry in Ray Tracing

CS6965 Fall 2011

Programming Trax

- Need to be aware of:
  - Thread assignment (atomicinc)
  - Global memory
  - Special function units
  - Calling convention (use const &)
  - NO DOUBLES!
  - Inline everything (when possible)
  - No dynamic memory (new, malloc)

trax.hpp

- Header with Trax intrinsics
  - Memory Ops: load and store
  - Thread management: atomicinc, barrier, synch
  - Arithmetic: min, max, invsqrt
  - Debug: profile, trax_print
  - Other

trax.hpp memory

- int loadi(int base, int offset)
- float loadf(int base, int offset)
- void storei(int base, int offset)
- void storef(int base, int offset)

trax.hpp misc

- int atomicinc(int location)
  - location is a global register {return globals[location]++; }
- void profile(int prof_id)
  - Starts/stops profiling (toggle)
- void trax_printi(int value)
- void trax_printf(float value)

trax.hpp math

- float min(float left, float right)
- float max(float left, float right)
- float invsqrt(float value)
- float sqrt(float value)
  - Just calls invsqrt

trax.hpp helpers

- Helper functions
  - trax_setup() – sets up the global memory to look like trax
  - trax_cleanup() – writes the image to a file
  - GetXRes(), GetYRes() – Loads the resolution from global memory
  - GetFrameBuffer() – Loads the address of the frame buffer from global memory

Compiling

- Two versions of the code are built
  - trax_cpp compiles to a native executable
  - trax_trax compiles to trax assembly
  - Makefile ensures they both deal with the same memory layout
- Running in SimHWRT
  - Use scripts/example.sh as an example of how to run simhwrt

Debugging

- For the executable version, TRAX=0
- For the trax version, TRAX=1
- Use this to do standard printf only when TRAX=0
  - #if TRAX==0
  - #include <stdio.h>
  - #endif

Questions?

Color

- Red-Green-Blue
- Forget other color spaces (for now)
- (Red, Green, Blue)
- Range of values: [0, 1.0] (maybe higher)

Color Math

- Given RGBs a, b and scalar k
  - $+a = (a_r, a_g, a_b)$
  - $-a = (-a_r, -a_g, -a_b)$
  - $k * a = a * k = (ka_r, ka_g, ka_b)$
  - $a/k = (a_r/k, a_g/k, a_b/k)$
  - $a \pm b = (a_r \pm b_r, a_g \pm b_g, a_b \pm b_b)$
  - $a * b = (a_r b_r, a_g b_g, a_b b_b)$
  - $a/b = (a_r/b_r, a_g/b_g, a_b/b_b)$

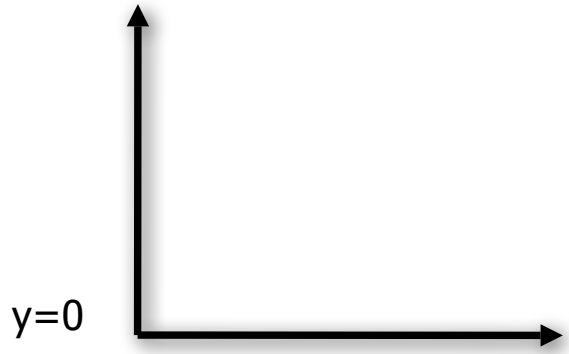
What to Implement

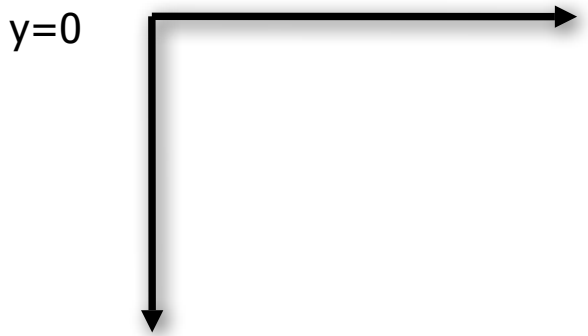
- Colors represented as float
- Define operators that make sense:
  - Color * Color
  - Color * scalar
  - Color ± Color
- Don't go overboard
- Colors could be a vector class, but...

Image

- An 2D array of Pixels
- Pixels can be light weight color
  - Our image output clamps to 0-255 from float framebuffer
- Stored in global memory (in trax)
- Global writes
  - storei() storef() intrinsics

Image gotchas





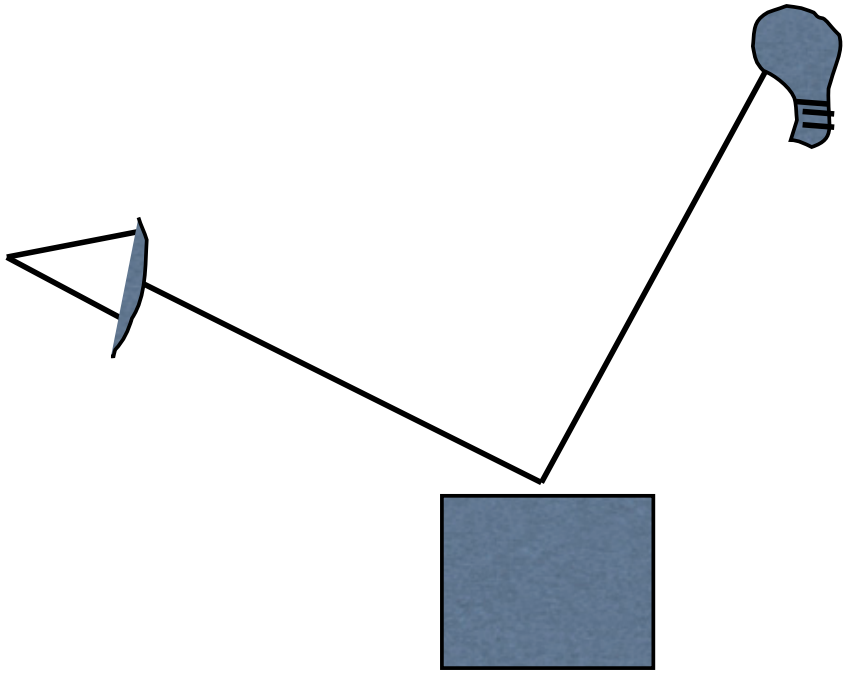
- Be careful - image coordinate system is "upside down"

Real world
Our ray tracer
OpenGL
Taught since 2nd grade

Televisions
Raster Images
Other 1950's technology

Ray tracing

- Ray tracing: Following paths of light to an imaginary camera



Geometry for graphics

- If it doesn't have any math, it probably isn't worth doing
- It it has too much math, it definitely isn't worth doing

Geometry for graphics

- What types of geometric queries would be useful to perform?
  - Given a line and a point, which side of the line does the point lie on?
  - Given two lines, do they intersect and where?
  - Others?

Geometric entities for ray tracing

- Point
- Vector
- Triangle
- Line
- Line segment
- Ray
- Plane
- Hyperplane

Vector space

- If V is a vector space, then the following are true ($\vec{X}, \vec{Y}, \vec{Z}$ are vectors, r, s are scalars):
- Commutativity:

$$\vec{X} + \vec{Y} = \vec{Y} + \vec{X}$$

- Associativity of vector addition:

$$(\vec{X} + \vec{Y}) + \vec{Z} = \vec{X} + (\vec{Y} + \vec{Z})$$

- Additive Identity:

$$\vec{X} + \vec{0} = \vec{0} + \vec{X} = \vec{X}$$

Vector space, continued

- Existence of additive inverse:

$$\vec{X} + (-\vec{X}) = 0$$

- Associativity of scalar multiplication:

$$(rs)\vec{X} = r(s\vec{X})$$

- Distributivity of scalar sums:

$$(r + s)\vec{X} = r\vec{X} + s\vec{X}$$

- Distributivity of vector sums:

$$r + (\vec{X} + \vec{Y}) = r\vec{X} + r\vec{Y}$$

- Scalar multiplication identity:

$$1\vec{X} = \vec{X}$$

Euclidean space

- A Euclidean space is a vector space where the vector is three real numbers: $\mathfrak{R}3$

Geometries

- Plane Geometry
- Parabolic Geometry
- Hyperbolic Geometry
- Euclidean Geometry
- Elliptic Geometry

Euclidean geometry

- Euclidean geometry defines 5 postulates:
- A straight line segment can be drawn joining any two points.
- Any straight line segment can be extended indefinitely in a straight line.
- Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as center.
- All right angles are congruent.
- If two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines inevitably must intersect each other on that side if extended far enough.

Another geometry

- What about projective geometry?

Affine space

- An affine space defines points with these properties (P,Q are points, A,B are vectors):

$$\vec{P} + \vec{0} = \vec{P}$$

$$\vec{P} + (\vec{A} + B) = (\vec{P} + \vec{A}) + \vec{B}$$

For any $\vec{Q}$, there exists $\vec{A}$ such that:

$$\vec{Q} = \vec{P} + \vec{A}$$

$$(\vec{A} = \vec{P} - \vec{Q})$$

Points and Vectors

- Why did we define points differently than vectors?

- Definition of linear transform T:

$$T(A+B) = T(A) + T(B)$$

$$sT(A) = T(sA)$$

Affine transformations

- An affine transformation preserves colinearity and ratios of distances
- Projective transform (ala OpenGL) is not Affine
- Rotation, scale, shear, translation, reflection are all affine

Affine transformations

$$M\vec{p} + \vec{T} = \vec{p}'$$

- (where M is a 3x3 transform and T is a translation vector)

- Can also be written as a 4x3 matrix:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & T_X \\ M_{10} & M_{11} & M_{12} & T_Y \\ M_{20} & M_{21} & M_{22} & T_Z \end{bmatrix}$$

- Affine transformations of points:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & T_x \\ M_{10} & M_{11} & M_{12} & T_y \\ M_{20} & M_{21} & M_{22} & T_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix}$$

- Affine transformations of vectors:

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & T_x \\ M_{10} & M_{11} & M_{12} & T_y \\ M_{20} & M_{21} & M_{22} & T_z \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ V_z \\ 0 \end{bmatrix} = \begin{bmatrix} V'_x \\ V'_y \\ V'_z \\ 0 \end{bmatrix}$$

Programming note

- There are at least three ways of distinguishing points and vectors:
  - Make separate Point and Vector classes
  - Make a single Vector class and store four values with 0 or 1 in the fourth component
  - Have two methods for transformation, one for points and one for vectors

Geometric consistency

- We defined axioms for adding vectors but what is the geometric meaning of P1 + P2?
- What about s*P1?

Interpolation

- What about finding the midpoint of a line between two points?
- $$P_{mid} = 0.5\ P1 + 0.5\ P2$$
- This is called an affine combination, and is valid for an arbitrary number of points only if the weights sum to 1

Affine transformations

- Affine transformations are also called Affine maps
- They have a few properties:
- An Affine transform is linear when applied to vectors:
  - T(A+B) = T(A)+T(B)
  - T(sA) = sT(A)
- An Affine transform preserves the plus operator for points/vectors
  - T(P+V) = T(P)+T(V)
- Affine transformations can be composed

Affine transformations

- Projective transform (ala OpenGL) is not Affine
- Rotation, scale, shear, translation, reflection are all affine
- Affine is a super-set of linear

Implementation

- Recommendation: Don't define multiplication on points
- Interpolate(float, Point, Point)
- AffineCombination(float, Point, float, Point)
- AffineCombination(float, Point, float, Point, float, Point)

Operators

- Operators that do make sense:
- P = P+V
- P = P-V
- V = P-P
- V = V + V
- V = V - V
- V = s * V
- V = - V

Cheating

- Sometimes all this religion forces you to be inefficient
- This will allow you to work around it if necessary:
  - explicit Point(const Vector&)
  - explicit Vector(const Point&)
  - And/or:
  - Point Vector::asPoint()
  - Vector Point::asVector()
- Also define V*V and a way to access each component

A single vec class

- Danny says you should use a single vec class
- I'll leave it up to you if you make separate vec or point classes
- Be aware of the tradeoffs

Vector operations

- There are a few other operations that we will need: first, an inner product
- Inner product: a way to multiply vectors resulting in a scalar with these properties:

$$\langle \vec{A} + \vec{B}, \vec{C} \rangle = \langle \vec{A}, \vec{B} \rangle + \langle \vec{A}, \vec{C} \rangle$$

$$\langle s\vec{A}, \vec{B} \rangle = s\langle \vec{A}, \vec{B} \rangle$$

$$\langle \vec{A}, \vec{B} \rangle = \langle \vec{B}, \vec{A} \rangle$$

$$\langle \vec{A}, \vec{A} \rangle \geq 0 \text{ and } \langle \vec{A}, \vec{A} \rangle = 0 \textit{ iff } \vec{A} = \vec{0}$$

More spaces

- Eulerian space is an inner product space
- More specifically, a Hilbert space

Dot product

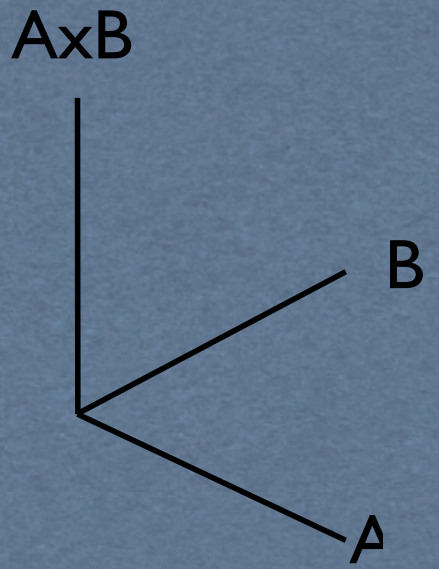
- The inner product for Eulerian space is called the Dot product:
- $A \cdot B = AxBx + AyBy + AzBz$
- Useful properties:
  - $A \cdot B = |A|\,|B| \cos \theta$
  - Invariant under rotation
  - is commutative, distributive, and associative

Distances

- A ·A is "magnitude" or length squared
- The distance between P1 and P2 is the length of the vector P2-P1
- If A ·A == 1, the vector is called a "unit vector" or is normalized
- Multiply a vector by the inverse of the length to find a unit vector having the same direction

Cross product

$$\vec{C} = \vec{A} \times \vec{B}$$



Can be written as a determinant:

$$\begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ A_x & A_y & A_z \\ B_x & B_y & C_z \end{vmatrix}$$

Expanded:

$$\left[A_y B_z + A_z B_y, A_z B_x + A_x B_z, A_x B_y + A_y B_x \right]$$

Cross product properties

$$\left|\vec{A} \times \vec{B}\right| = \left|\vec{A}\right| \sin\theta$$

$$\vec{A} \times \vec{B} = -\vec{B} \times \vec{A}$$

$$\vec{A} \times (\vec{B} + \vec{C}) = (\vec{A} \times \vec{B}) + (\vec{A} \times \vec{C})$$

$$(s\vec{A}) \times \vec{B} = s(\vec{A} \times \vec{B})$$

$\left|\vec{A} \times \vec{B}\right|$ = Area of swept rectangle

If $\vec{C} = \vec{A} \times \vec{B}$, then $\vec{C}$ is perpendicular to A and B

$$(\vec{C} \cdot \vec{A} = \vec{C} \cdot \vec{B} = 0)$$

If $\vec{A} = \vec{B}$ or $\vec{A} = -\vec{B}$ then $\vec{A} \times \vec{B} = \vec{0}$

Scalar triple product

- A ·(BxC) = B ·(CxA) = C ·(AxB)
- Can also be written as the determinant:

$$\begin{vmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{vmatrix}$$

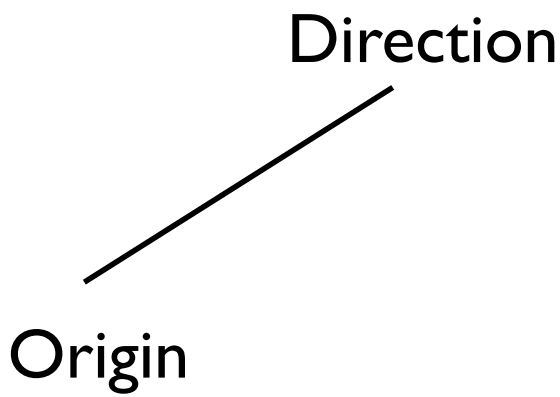
- This one is pretty cool, but we won't use it often.

Implementation notes

- Dot product and cross product are awkward.
  - As a standalone function: Dot(A, B)
  - As a member method: A.dot(B)
- I don't recommend overloading ^ or some other obscure operator. Operator precedence will bite you and nobody will be able to read your code

Rays

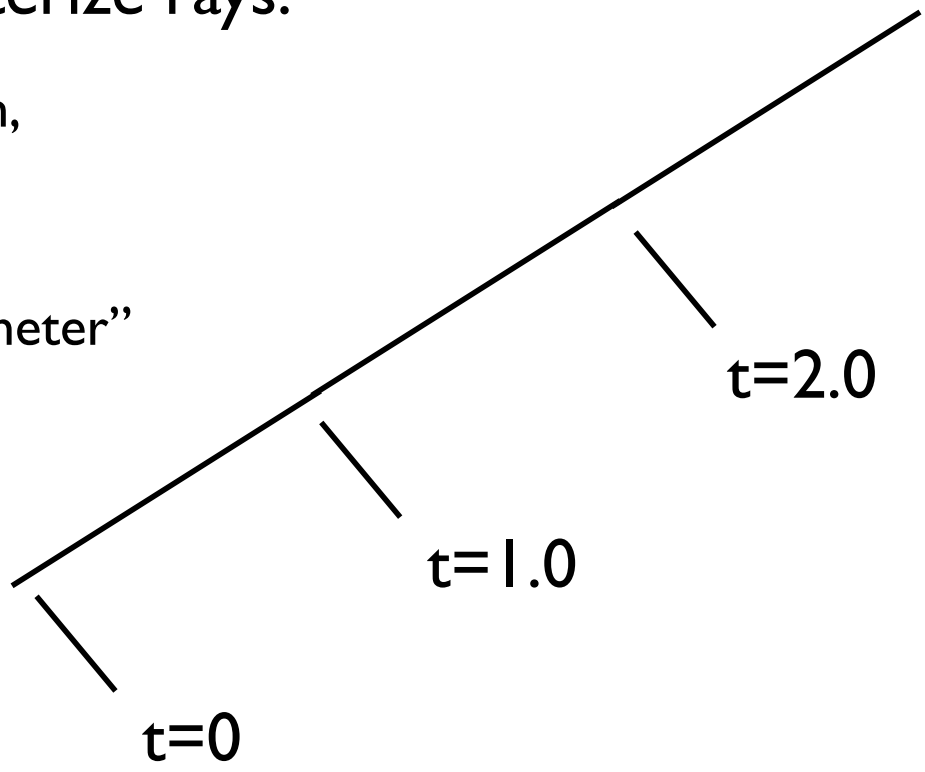
- A ray consists of a point and a vector:
- Class Ray {
- Point origin;
- Vector direction;
- …
- };



Parametric Rays

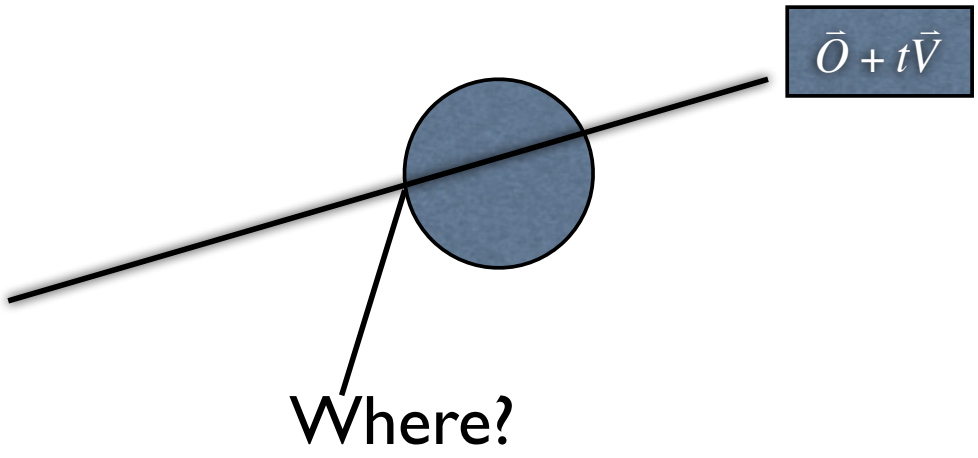
- We usually parameterize rays:
  - Where O is the origin,
  - V is direction,
  - and t is the "ray parameter"

$$\vec{P} = \vec{O} + t\vec{V}$$



Geometric Queries

- Back to the original question:
  - What queries can we perform on our virtual geometry?
- Ray tracing: determine if (and where) rays hit an object



Ray-sphere intersection

- What is the implicit equation for a sphere centered at the origin?

Ray-sphere intersection

- What is the implicit equation for a sphere centered at the origin?

$$x^2 + y^2 + z^2 - r^2 + 0$$

Ray-sphere intersection

Sphere: $x^2 + y^2 + z^2 - r^2 = 0$

Ray: $\left[O_x + tV_x, O_y + tV_y, O_z + tV_z\right]$

Ray-sphere intersection

Sphere: $x^2 + y^2 + z^2 - r^2 = 0$

Ray: $\left[O_x + tV_x, O_y + tV_y, O_z + tV_z\right]$

$$\left(O_x + tV_x\right)^2 + \left(O_y + tV_y\right)^2 + \left(O_z + tV_z\right)^2 - r^2 = 0$$

Ray-sphere intersection

Sphere: $x^2 + y^2 + z^2 - r^2 = 0$

Ray: $\left[O_x + tV_x, O_y + tV_y, O_z + tV_z\right]$

$$\left(O_x + tV_x\right)^2 + \left(O_y + tV_y\right)^2 + \left(O_z + tV_z\right)^2 - r^2 = 0$$

$$O_x^{\,2} + 2tV_x + t^2V_x^{\,2} + O_y^{\,2} + 2tV_y + t^2V_y^{\,2} + O_z^{\,2} + 2tV_z + t^2V_z^{\,2} - r^2 = 0$$

Ray-sphere intersection

Sphere: $x^2 + y^2 + z^2 - r^2 = 0$

Ray: $\left[O_x + tV_x, O_y + tV_y, O_z + tV_z\right]$

$$\left(O_x + tV_x\right)^2 + \left(O_y + tV_y\right)^2 + \left(O_z + tV_z\right)^2 - r^2 = 0$$

$$O_x^{\,2} + 2tV_x + t^2V_x^{\,2} + O_y^{\,2} + 2tV_y + t^2V_y^{\,2} + O_z^{\,2} + 2tV_z + t^2V_z^{\,2} - r^2 = 0$$

$$O_x^{\,2} + O_y^{\,2} + O_z^{\,2} + 2tV_x + 2tV_y + 2tV_z + t^2V_x^{\,2} + t^2V_y^{\,2} + t^2V_z^{\,2} - r^2 = 0$$

Ray-sphere intersection

Sphere: $x^2 + y^2 + z^2 - r^2 = 0$

Ray: $\left[O_x + tV_x, O_y + tV_y, O_z + tV_z\right]$

$$\left(O_x + tV_x\right)^2 + \left(O_y + tV_y\right)^2 + \left(O_z + tV_z\right)^2 - r^2 = 0$$

$$O_x^2 + 2tV_x + t^2V_x^2 + O_y^2 + 2tV_y + t^2V_y^2 + O_z^2 + 2tV_z + t^2V_z^2 - r^2 = 0$$

$$O_x^2 + O_y^2 + O_z^2 + 2tV_x + 2tV_y + 2tV_z + t^2V_x^2 + t^2V_y^2 + t^2V_z^2 - r^2 = 0$$

$$t^2\left(V_x^2 + V_y^2 + V_z^2\right) + 2t\left(V_x + V_y + V_z\right) + O_x^2 + O_y^2 + O_z^2 - r^2 = 0$$

Ray-sphere intersection

$$t^2\left(V_x^2 + V_y^2 + V_z^2\right) + 2t\left(V_x + V_y + V_z\right) + O_x^2 + O_y^2 + O_z^2 - r^2 = 0$$

A quadratic equation, with

$$a = V_x^2 + V_y^2 + V_z^2$$

$$b = 2\left(V_x + V_y + V_z\right)$$

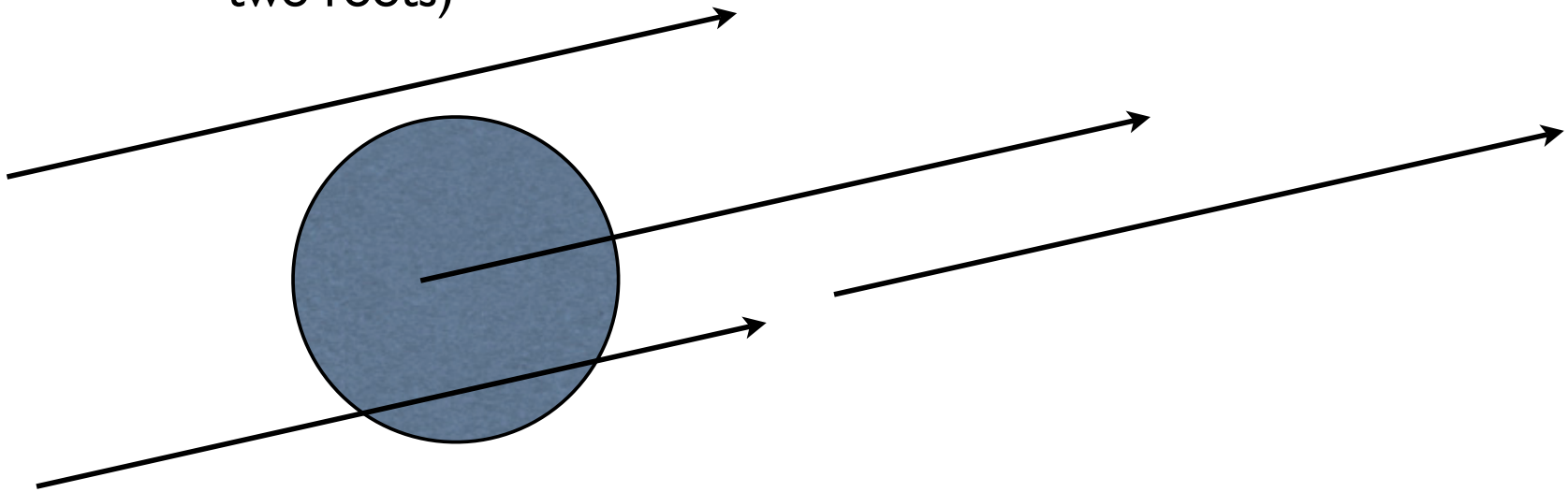
$$c = O_x^2 + O_y^2 + O_z^2 - r^2$$

roots:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}, \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Ray-sphere intersection

- If the discriminant $\left(b^2 - 4ac\right)$ is negative, the ray misses the sphere
  - Otherwise, there are two distinct intersection points (the two roots)



Ray-sphere intersection

- What about spheres not at the origin?
- For center C, the equation is:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$

- We could work this out, but there must be an easier way…

Ray-sphere intersection, improved

- Points on a sphere are equidistant from the center of the sphere
- Our measure of distance: dot product
- Equation for sphere:

$$\left(\vec{P} - \vec{C}\right) \bullet \left(\vec{P} - \vec{C}\right) - r^2 = 0$$

Ray-sphere intersection, improved

- Points on a sphere are equidistant from the center of the sphere
- Equation for sphere:

$$\left(\vec{P}-\vec{C}\right)\bullet\left(\vec{P}-\vec{C}\right)-r^2=0$$

$$\vec{P}=\vec{O}+t\vec{V}$$

$$\left(\vec{O}=t\vec{V}-\vec{C}\right)\bullet\left(\vec{O}=t\vec{V}-\vec{C}\right)-r^2=0$$

$$t^2\vec{V}\bullet\vec{V}+2t\left(\vec{O}-\vec{C}\right)\bullet\vec{V}+\left(\vec{O}-\vec{C}\right)\bullet\left(\vec{O}-\vec{C}\right)-r^2=0$$

Ray-sphere intersection, improved

$$t^2\vec{V}\bullet\vec{V} + 2t\left(\vec{O}-\vec{C}\right)\bullet\vec{V} + \left(\vec{O}-\vec{C}\right)\bullet\left(\vec{O}-\vec{C}\right) - r^2 = 0$$

Vector $\vec{O}' = \vec{O} - \vec{C}$

$a = \vec{V}\bullet\vec{V}$

$b = 2\vec{O}'\bullet\vec{V}$

$c = \vec{O}'\bullet\vec{O}' - r^2$

Program 1

- Implement Point, Vector, Color, Image, and Sphere classes
- Sphere class is a throwaway prototype, but the others you will use forever
- Creative image: Rearrange the same spheres (or make new ones)