UMD CS Computer Graphics

Distribution Ray Tracing

Pete Willemsen

University of Minnesota Duluth

April 16, 2013

Outline

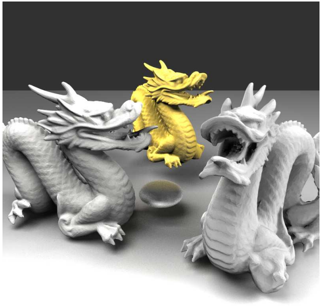
Objective

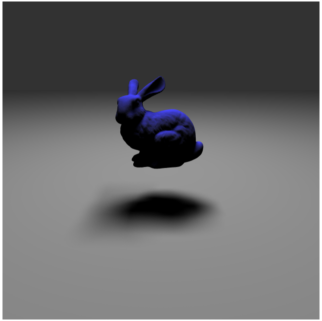
Develop a Cook-style ray tracer that provides more appropriate sampling strategies with an accelerated intersection data structure.

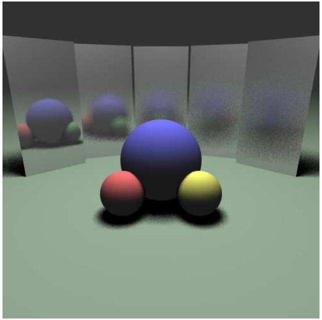
Outcomes for this section of course

- Instancing of renderable objects
- Spatial data structures for accelerating ray tracers
- Loading more complicated objects, such as triangle meshes
- Develop better sampling mechanisms for ray tracing effects

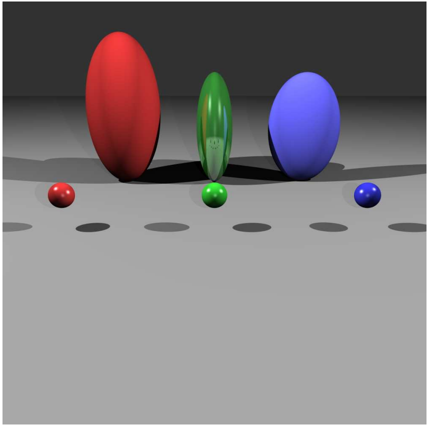
Examples





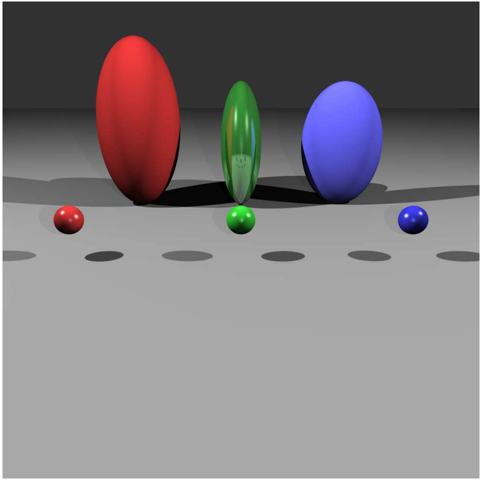


Object instancing



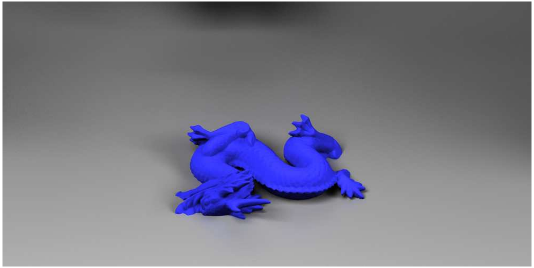
- Conserve memory footprint - reuse object geometry
- Create interesting scenes - Manipulate and transform objects

Focus on ellipsoid spheres



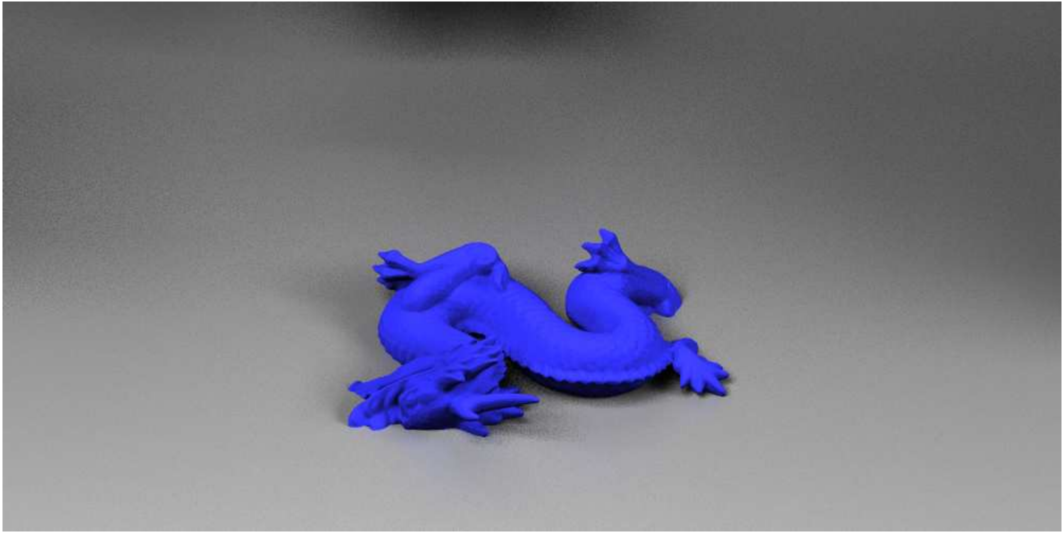
How can these be produced?

Another example to consider:



What is the blue dragon's local coordinate system? (*Hint: You can assume Cartesian Coordinates are as I'd draw them on the board*)

Rotate object about X axis by -90 degrees



Rotate object about X axis by -90 degrees

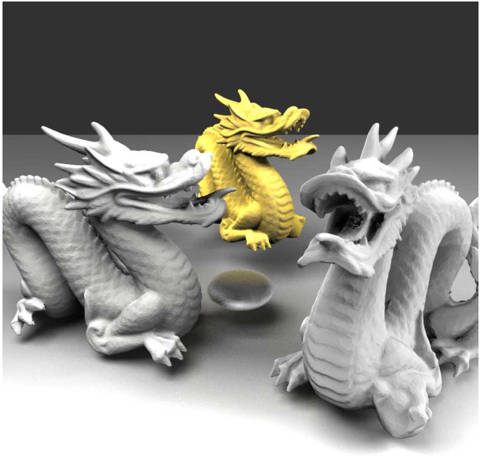
Rotate object about X axis by -90 degrees

Translate in +Y

Translate in +Y

Translate in +Y

Translate in +Y



How about other dragons?

- Perform additional transformations to create other instances of dragons
- Scale, translate, rotate

Object Instancing Requirements

- New class in renderable object hierarchy - *InstancedObject*
- Support for matrix transformations of object geometry

Object Instancing Requirements

class InstancedObject

- New class understands instancing and transformations
- Maintains a pointer to the base renderable object that was loaded only once (!!)
- Contains a matrix that describes how to transform the base object geometry
- Works seamlessly with other objects and ray tracer codebase

Object Instancing Requirements

class InstancedObject

- New class understands instancing and transformations
- Maintains a pointer to the base renderable object that was loaded only once (!!)
- Contains a matrix that describes how to transform the base object geometry
- Works seamlessly with other objects and ray tracer codebase

- Will need a new list of *special* objects so that the base models are only loaded once!

3D Linear Transformations

We will focus on 3D transformations:

- Scale - s_x, s_y, s_z - change the size of an object
- Rotate-Z - θ - rotate about the Z axis
- Rotate-Y - θ - rotate about the Y axis
- Rotate-X - θ - rotate about the X axis

3D Linear Transformations

$$\text{Scale}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

3D Linear Transformations

Rotations

$$\text{Rotate-X}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

$$\text{Rotate-Y}(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

$$\text{Rotate-Z}(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

What properties do these matrices all have?

3D Linear Transformations

Rotations

$$\text{Rotate-X}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

$$\text{Rotate-Y}(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

$$\text{Rotate-Z}(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

What properties do these matrices all have?

- Rows are orthogonal to each other
- Columns are orthogonal to each other
- Represent the basis vectors of some rotation

Arbitrary Rotations

General form for all rotations:

$$R_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$

These are the components of a set of orthogonal (basis) vectors!

- $\vec{u} = x_u\vec{x} + y_u\vec{y} + z_u\vec{z}$
- $\vec{v} = x_v\vec{y} + y_v\vec{y} + z_v\vec{z}$
- $\vec{w} = x_w\vec{z} + y_w\vec{y} + z_w\vec{z}$

So, with basis vectors being orthogonal,

- $\vec{u} \cdot \vec{u} = \vec{v} \cdot \vec{v} = \vec{w} \cdot \vec{w} = 1$
- $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{u} = 0$

Arbitrary Rotation Matrices - Meaning

So, what does that mean. Start by multiplying $\vec{u}$ by R_{uvw}. What happens?

$$R_{uvw}\vec{u} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix} \begin{bmatrix} x_u \\ y_u \\ z_u \end{bmatrix} = \begin{bmatrix} x_u x_u + y_u y_u + z_u z_u \\ x_v x_u + y_v y_u + z_v z_u \\ x_w x_u + y_w y_u + z_w z_u \end{bmatrix}$$

Well, for starters, $R_{uvw}\vec{u}$ is really the dot product between the rows and $\vec{u}$:

$$R_{uvw}\vec{u} = \begin{bmatrix} \vec{u} \cdot \vec{u} \\ \vec{v} \cdot \vec{u} \\ \vec{w} \cdot \vec{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \vec{x}$$

Similarly, $R_{uvw}\vec{v} = \vec{y}$ and $R_{uvw}\vec{w} = \vec{z}$!

Arbitrary Rotation Matrices - Meaning

Thus,

- R_{uvw} takes the basis $\vec{u}\vec{v}\vec{w}$ to the Cartesian coordinate system via a rotation operation.

How do you go back from the Cartesian coordinate system to the $\vec{u}\vec{v}\vec{w}$ basis?

Arbitrary Rotation Matrices - Meaning

Thus,

- R_{uvw} takes the basis $\vec{u}\vec{v}\vec{w}$ to the Cartesian coordinate system via a rotation operation.

How do you go back from the Cartesian coordinate system to the $\vec{u}\vec{v}\vec{w}$ basis?

By the inverse of R_{uvw}

Arbitrary Rotations - From XYZ to UVW

To go from the Cartesian coordinate system to the uvw system, we use R_{uvw}^{-1}

Inverse of R_{uvw} is R_{uvw}^{T}, or the transpose of R_{uvw}

- If R_{uvw} is a rotation matrix with orthogonal rows, the R_{uvw}^{T} is a rotation matrix with orthogonal columns
- Inverse of an orthogonal matrix is always its transpose
- And, R_{uvw}^{T} is in fact R_{uvw}^{-1}

Thus,

$$R_{uvw}^{T}\vec{x} = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_u \\ y_u \\ z_u \end{bmatrix}$$

Similarly, $R_{uvw}^{T}\vec{y} = \vec{v}$ and $R_{uvw}^{T}\vec{z} = \vec{w}$!

Rotating about Arbitrary Axis

You now have enough foundation to rotate an object about an arbitrary axis not just the Cartesian axis!

1. You must create an orthonormal basis about the arbitrary axis (you know how to do that!)
2. Rotate the basis to the Cartesian coordinate system, R_{uvw}
3. Apply your rotation about the Z-axis
4. Rotate back to the the original basis frame from the Cartesian coordinate system, R_{uvw}^T

$$M = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$

Affine Transformations and Translation

Thus far, all discussion has been with transforms that have the form:

$$x^{'} = a_{11}x + a_{12}y$$

$$y^{'} = a_{21}x + a_{22}y$$

These transforms can only scale and rotate objects and cannot move them!

What we need is to be able to perform

$$x^{'} = x + x_t$$

$$y^{'} = y + y_t$$

However, it is not possible to add that translation to a 2x2 matrix!

Affine Transformations and Translation

To achieve what we want, we will use 3x3 matrices (for 2D transformations) and represent point (x, y) as $[xy1]^T$.

$$\begin{bmatrix} a_{11} & a_{12} & x_t \\ a_{21} & a_{22} & y_t \\ 0 & 0 & 1 \end{bmatrix}$$

All vectors must now have a 1 in the last place!

$$\begin{bmatrix} x_\prime \\ y_\prime \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & x_t \\ a_{21} & a_{22} & y_t \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + x_t \\ a_{21}x + a_{22}y + y_t \\ 1 \end{bmatrix}$$

These are called *affine transformations* using *homogeneous coordinates*!

Affine Transformations

Some issues:

- Transformations were for points!
- What about offsets, displacements, or directions?

For locations, last coordinate will be 1.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

For directions, last coordinate will be 0.

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Affine Transformations in 3D

Works fine in 3D:

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ z + z_t \\ 1 \end{bmatrix}$$

Homogeneous Transform

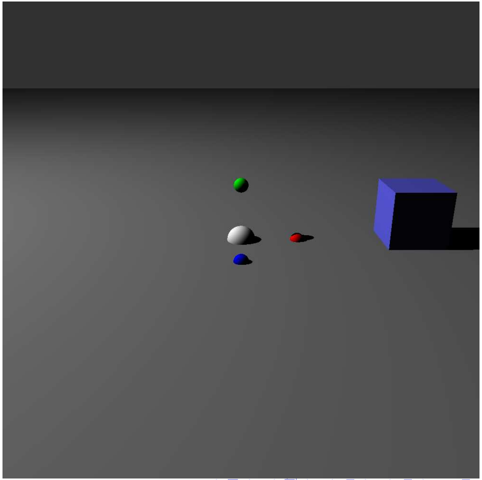
This allows us to define a 4x4 Matrix that holds a rotation and a translation (with rotation happening first):

$$\begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & x_t \\ a_{21} & a_{22} & a_{23} & y_t \\ a_{31} & a_{32} & a_{33} & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Examples

Let's consider order of operations. First, translate a box with the definition $minPt = (-0.5, -0.5, -0.5)$ and $maxPt = (0.5, 0.5, 0.5)$.

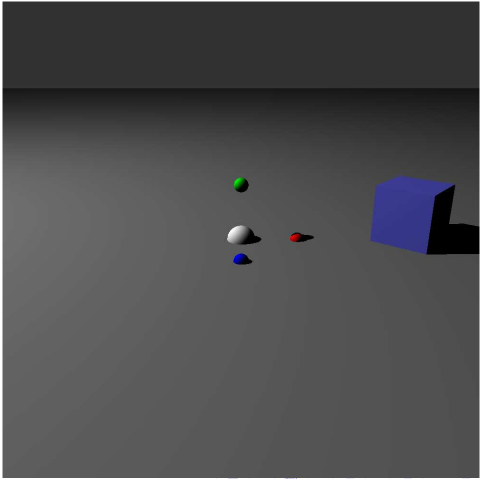
$$M_{translate} = \begin{bmatrix} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Examples

Next, what happens if a rotation preceeds the translation?

$$M = \begin{bmatrix} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(50) & 0 & \sin(50) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(50) & 0 & \cos(50) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

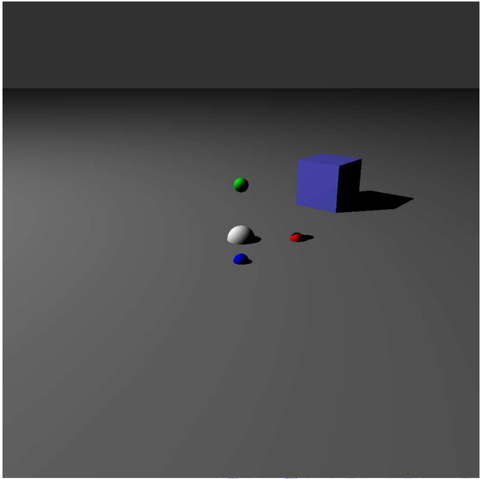


Examples

What happens if these matrices are reversed and the translation preceeds rotation?

$$M = \begin{bmatrix} \cos(50) & 0 & \sin(50) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(50) & 0 & \cos(50) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The order of operation matters immensely! In the matrix order we present the right-most matrix occurs first.



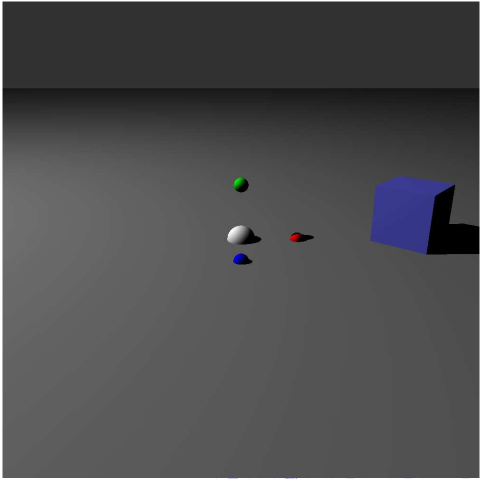
Examples

Now, back to the transformation where the rotation is first, followed by translation:

$$M = \begin{bmatrix} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(50) & 0 & \sin(50) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(50) & 0 & \cos(50) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.642788 & 0 & 0.766044 & 3 \\ 0 & 1 & 0 & 0.5 \\ -0.766044 & 0 & 0.642788 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is the *homogeneous transform*. It combines two matrices: a rotation matrix and a translation matrix such that the rotation matrix is applied first.



Examples

Try it. Multiply the *maxPt = (0.5, 0.5, 0.5)* first by the rotation matrix and then by the translation matrix. What's the output?

$$M_{translate} = \begin{bmatrix} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{rotate} = \begin{bmatrix} 0.642788 & 0 & 0.766044 & 0 \\ 0 & 1 & 0 & 0 \\ -0.766044 & 0 & 0.642788 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Next, multiple the *maxPt* by first homogeneous transform matrix. What's the output?

$$M = \begin{bmatrix} 0.642788 & 0 & 0.766044 & 3 \\ 0 & 1 & 0 & 0.5 \\ -0.766044 & 0 & 0.642788 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider other transformations

A rotation, followed by a scale, followed by another rotation, followed by translation:

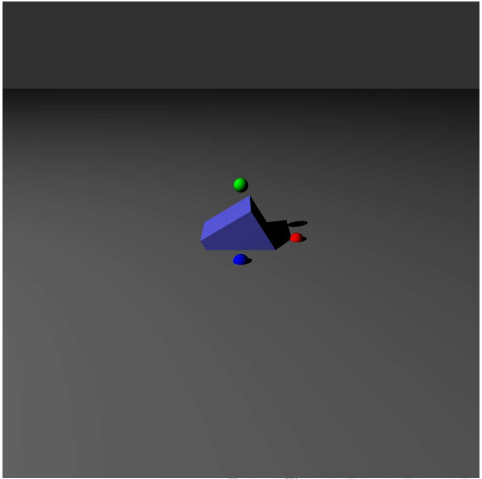
```
<transform name="xform1">
  <translate>3.0 0.5 0.0</translate>
  <rotate axis="Z">−58.3</rotate>
  <scale>1.618 0.618 1</scale>
  <rotate axis="Z">31.7</rotate>
</transform>
```

We'll look at the scenes as each step is applied:

Consider other transformations

A rotation:

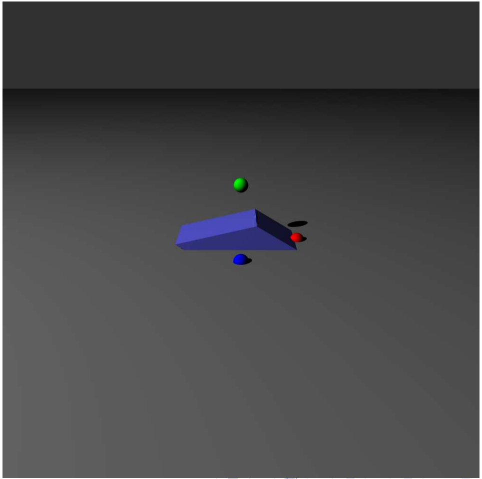
```
    <transform name="xform1">
<!--      <translate>3.0 0.5 0.0</translate>
      <rotate axis="Z">-58.3</rotate>
      <scale>1.618 0.618 1</scale> -->
      <rotate axis="Z">31.7</rotate>
    </transform>
```



Consider other transformations

A rotation, followed by a scale:

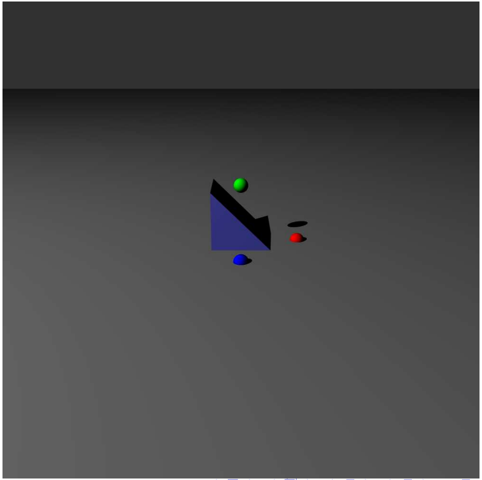
```
    <transform name="xform1">
<!--      <translate>3.0 0.5 0.0</translate>
      <rotate axis="Z">-58.3</rotate> -->
      <scale>1.618 0.618 1</scale>
      <rotate axis="Z">31.7</rotate>
    </transform>
```



Consider other transformations

A rotation, followed by a scale, followed by another rotation:

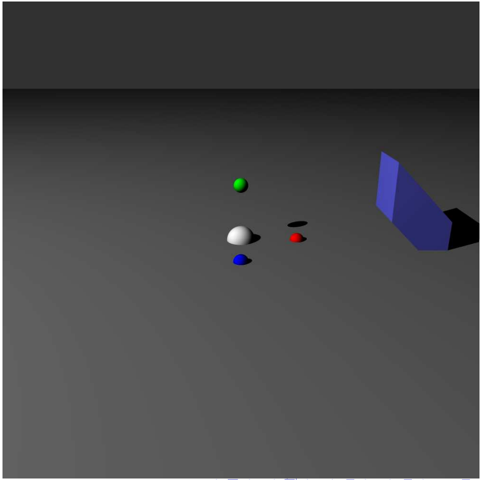
```
    <transform name="xform1">
<!--        <translate>3.0 0.5 0.0</translate> -->
      <rotate axis="Z">-58.3</rotate>
      <scale>1.618 0.618 1</scale>
      <rotate axis="Z">31.7</rotate>
    </transform>
```



Consider other transformations

A rotation, followed by a scale, followed by another rotation, followed by translation:

```
<transform name="xform1">
  <translate>3.0 0.5 0.0</translate>
  <rotate axis="Z">-58.3</rotate>
  <scale>1.618 0.618 1</scale>
  <rotate axis="Z">31.7</rotate>
</transform>
```



Consider Inverse Actions

How do you *undo* a matrix transformation?

- Apply the inverse operation

For instance, a translation:

$$M_{translate} = \begin{bmatrix} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

can be undone by the opposite translation:

$$M_{translate'} = \begin{bmatrix} 1 & 0 & 0 & -3.0 \\ 0 & 1 & 0 & -0.5 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider Inverse Actions

How do you *undo* a matrix transformation?

- Apply the inverse operation

For instance, a rotation:

$$M_{rotZ} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & -\cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

can be undone by the opposite rotation:

$$M_{rotZ'} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 & 0 \\ \sin(-\theta) & -\cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider Inverse Actions

Look at rotation a bit more with theta = 45.0:

$$M_{rotZ} = \begin{bmatrix} 0.7071 & -0.7071 & 0 & 0 \\ 0.7071 & -0.7071 & 0 & 0 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is undone by the opposite rotation of -45.0:

$$M_{rotZ'} = \begin{bmatrix} 0.7071 & 0.7071 & 0 & 0 \\ -0.7071 & -0.7071 & 0 & 0 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

But,

$$M_{rotZ}^{transpose} = \begin{bmatrix} 0.7071 & 0.7071 & 0 & 0 \\ -0.7071 & -0.7071 & 0 & 0 \\ 0 & 0 & 1 & 0.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider Inverse Actions

Generally, undoing a matrix transformation is as simple as applying the inverse matrix transformation. Thus, you will need support to compute the matrix inverse of a homogeneous transform matrix:

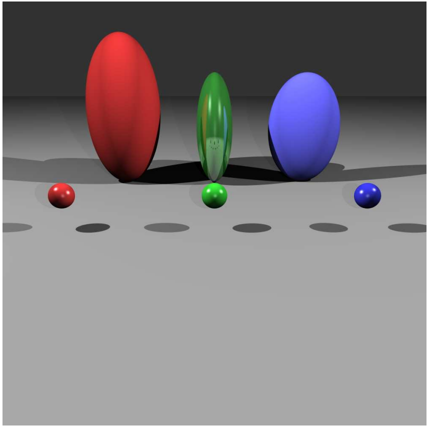
$$M^{-1} = inverse(M)$$

Then, with the inverse matrix, you can undo both the rotations, scales, and translations that were stored within the original homogeneous transform. Note! The transpose is not the inverse for homogeneous transform matrices. This rule is *only* true for pure rotation matrices! You must implement the determinant calculations as specified in the text book around page 101.

Instancing Overview

Benefits of Instancing Objects

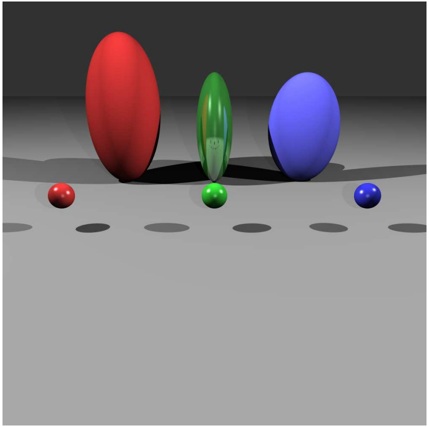
- Achieves object complexity by manipulating base objects
- Saves memory footprint by re-using objects as necessary
- Forces your code to deal with transformations



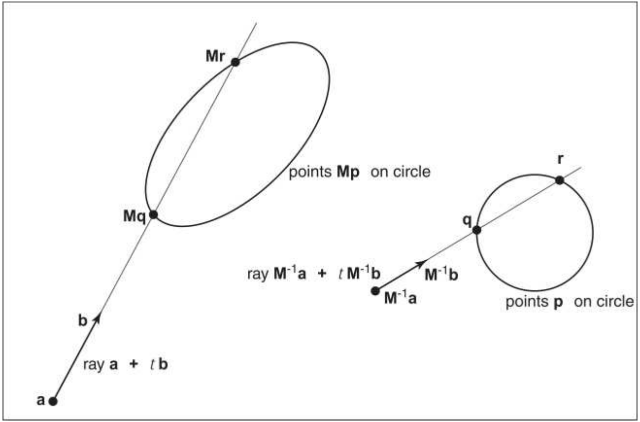
Instancing Overview

How can we achieve instancing?

- We have a single matrix that encompasses all transformations!
- Why not multiply all vertices with the homogeneous transform and move the object?
- How will you re-use this transformed object?



Instancing Overview



Key Idea: Transform the rays, not the objects! n

Instancing Algorithm

InstanceObject::intersect(Ray r, t_{min}, t_{max}, hitRecord rec)

Ray $r' \leftarrow \mathbf{M}^{-1} r.\vec{origin} + t\mathbf{M}^{-1} r.\vec{direction}$

if (baseObjectPtr→intersect(r', t_{min}, t_{max}, rec)) **then**

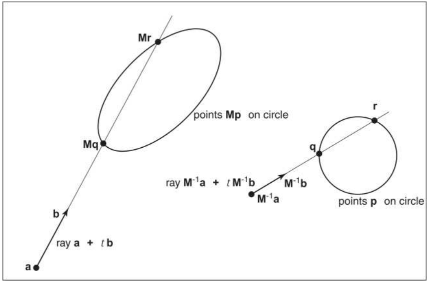
  rec.$\vec{n} \leftarrow (\mathbf{M}^{-1})^T$ rec.$\vec{n}$

  return true;

else

  return false;

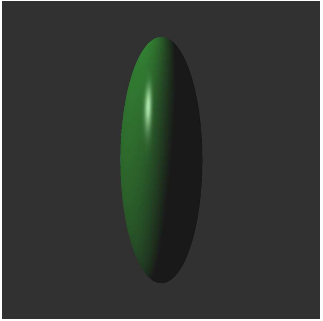
end if



Instancing Example

Consider the following simple example in which the unit sphere is instanced into an ellipsoid.

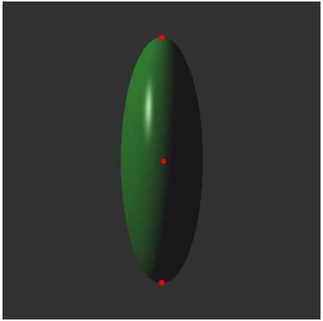
```
<instance name="unitSphere" type="sphere">
  <shader ref="blueMat"/>
  <center>0.0 0.0 0.0</center>
  <radius>1.0</radius>
</instance>
...
<shape name="ellipsoid" type="instance" id="unitSphere">
  <shader ref="greenMirroredBlinnPhong"/>
  <transform name="xform1">
    <translate>0.0 1.5 -4.0</translate>
    <scale>0.5 1.5 1.0</scale>
  </transform>
</shape>
```



Instancing Example

With this example, let's consider three rays from the camera that go through the very top of the ellipsoid, the middle, and the bottom of the ellipsoid. The coordinate below are world coordinates from the ray generation stage:

- Ray Origin: $(0, 1.5, 0)^T$
- Ray Direction (top): $(0, 0.192, -0.5)^T$
- Ray Direction (middle): $(0, 0, -0.5)^T$
- Ray Direction (bottom): $(0, -0.194, -0.5)^T$



Instancing Example

With this example, let's consider three rays from the camera that go through the very top of the ellipsoid, the middle, and the bottom of the ellipsoid. The coordinate below are world coordinates from the ray generation stage:

The Inverse matrix associated with the scale followed by translation of this transform follows:

- Ray Origin: $(0, 1.5, 0)$
- Ray Direction (top): $[0, 0.192, -0.5)^T$
- Ray Direction (middle): $[0, 0, -0.5)^T$
- Ray Direction (bottom): $[0, -0.194, -0.5)^T$

$$\mathbf{M} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 1.5 \\ 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0.66667 & 0 & -1 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Instancing Example - Tranforming the Ray

For the ray's origin, this evaluates to the following:

$$r'_{\vec{origin}} = M^{-1} r_{\vec{origin}}$$

$$\begin{bmatrix} 0 \\ 0 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0.66667 & 0 & -1 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1.5 \\ 0 \\ 1 \end{bmatrix}$$

Instancing Example - Tranforming the Ray

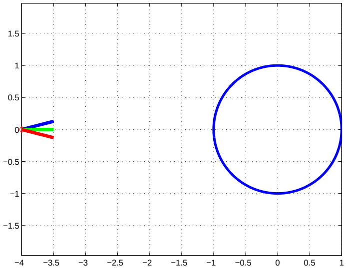
In the instanced object's intersection function, this data would transform the ray's origin along with these directions. The resulting operations follow:

- $\vec{r'_{origin}}^{T} = \mathbf{M}^{-1}[0 \quad 1.5 \quad 0 \quad 1]^T \rightarrow [0 \quad 0 \quad 4 \quad 1]^T$
- $r'_{dir_{Top}} = \mathbf{M}^{-1} \vec{r_{dir_{Top}}}^{T} \rightarrow [0 \quad 0.128 \quad -0.5 \quad 0]^T$
- $r'_{dir_{Mid}} = \mathbf{M}^{-1} \vec{r_{dir_{Mid}}}^{T} \rightarrow [0 \quad 0 \quad -0.5 \quad 0]^T$
- $r'_{dir_{Bot}} = \mathbf{M}^{-1} \vec{r_{dir_{Bot}}}^{T} \rightarrow [0 \quad -0.129 \quad -0.5 \quad 0]^T$

Instancing Example - Intersecting the Inverse Ray

Recall that these new rays (from the previous slide) are used to intersect the *unit sphere* that was the object that was instanced and transformed into the ellipsoid:

- $\vec{r'}_{origin}^{\;T} = [0 \quad 0 \quad 4 \quad 1]^T$
- $r'_{dir_{Top}} = [0 \quad 0.128 \quad -0.5 \quad 0]^T$
- $r'_{dir_{Mid}} = [0 \quad 0 \quad -0.5 \quad 0]^T$
- $r'_{dir_{Bot}} = [0 \quad -0.129 \quad -0.5 \quad 0]^T$



Instancing

Take note that

- Only requires adding a new class (InstancedObject) to handle the manipulations of the Rays
- Object pointers to the actual instanced objects are maintained (and likely reused amonst several InstancedObjects)

Take special note that

- The normal is transformed too!!
- Why?

Transforming Normal Vectors

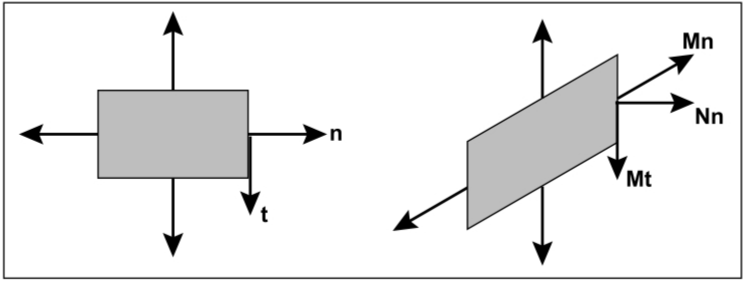
- What happens when you transform objects in the scene?

Transforming Normal Vectors

- What happens when you transform objects in the scene?
- Does it make sense to transform their normal vectors? What happens?

Transforming Normal Vectors

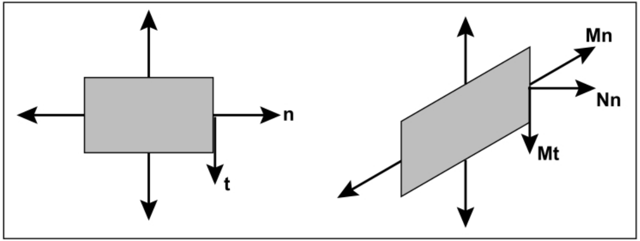
- What happens when you transform objects in the scene?
- Does it make sense to transform their normal vectors? What happens?
- Tranforming normal vectors with the transformation matrices:



- What's wrong?

Transforming Normal Vectors

We need to find a **N** such that normals are correctly transformed!



Need to maintain:

$$\vec{n}^T \vec{t} = 0$$

and need to find **N** such that

$$\vec{t}_M = \mathbf{M}\vec{t}$$

$$\vec{n}_N = \mathbf{N}\vec{n}$$

Transforming Normal Vectors

Recall that $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$, so

$$\vec{n}^T \vec{t}$$

$$= \vec{n}^T \mathbf{I} \vec{t}$$

$$= \vec{n}^T \mathbf{M}^{-1} \mathbf{M} \vec{t} = 0$$

Transforming Normal Vectors

Recall that $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$, so

$$\vec{n}^T\vec{t}$$
$$= \vec{n}^T\mathbf{I}\vec{t}$$
$$= \vec{n}^T\mathbf{M}^{-1}\mathbf{M}\vec{t} = 0$$

and now, let's try to see this as dot products:

$$(\vec{n}^T\mathbf{M}^{-1})(\mathbf{M}\vec{t}) = 0$$

Transforming Normal Vectors

Recall that $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$, so

$$\begin{aligned} & \vec{n}^T\vec{t} \\ &= \vec{n}^T\mathbf{I}\vec{t} \\ &= \vec{n}^T\mathbf{M}^{-1}\mathbf{M}\vec{t} = 0 \end{aligned}$$

and now, let's try to see this as dot products:

$$(\vec{n}^T\mathbf{M}^{-1})(\mathbf{M}\vec{t}) = 0$$

$$(\vec{n}^T\mathbf{M}^{-1})\vec{t}_M = 0$$

Transforming Normal Vectors

Thus, the left part of this expression is the vector that is perpendicular to $\vec{t}_M$:

$$(\vec{n}^T \mathbf{M}^{-1})\vec{t}_M = 0$$

This must be a row vector so we use the transpose to represent a column vector (our standard vectors) as a row vector:

$$\vec{n}_N^T = \vec{n}^T \mathbf{M}^{-1}$$

Transforming Normal Vectors

Thus, the left part of this expression is the vector that is perpendicular to $\vec{t}_M$:

$$(\vec{n}^T \mathbf{M}^{-1})\vec{t}_M = 0$$

This must be a row vector so we use the transpose to represent a column vector (our standard vectors) as a row vector:

$$\vec{n}_N^T = \vec{n}^T \mathbf{M}^{-1}$$

thus,

$$\vec{n}_N = (\vec{n}^T \mathbf{M}^{-1})^T$$
$$\vec{n}_N = (\mathbf{M}^{-1})^T \vec{n}$$

and so,

$$N = (\mathbf{M}^{-1})^T$$

Transforming Normal Vectors

Make sure that you tranform your normal vectors appropriately, when instancing objects!

$$N = (\mathbf{M}^{-1})^{T}$$