Computing the Pixel Coordinates of a 3D Point

Contents

Perspective Projection

Mathematics of Computing the 2D Coordinates of a 3D Point

Source Code

Finding the 2D pixel coordinates of a 3D Point Explained from Beginning to End

When a point or vertex is defined in the scene and is visible to the eye or to the camera, it appears in the image as a dot (or more precisely a pixel if the image is a digital one). We already talked about the perspective projection process which is used to convert the position of that point in 3D space to a position on the surface of the image. But this position is not expressed in terms of pixels coordinates. So how do we actually find the final 2D pixel coordinates of the projected point in the image? In this chapter, we will review the entire process by which points are converted from their original world position to their final raster position (their position in the image in terms of pixel coordinates).

> The technique we will describe in this lesson is specific to the rasterisation algorithm (the rendering technique used by GPUs to produce images of 3D scene). If you are interested to learn how it is done in ray-tracing, check the lesson Rays and Camera.

World Coordinate System and World Space

When a point is first defined in the scene, we say its coordinates are defined in **world space**: the coordinates of this point are defined with respect to a global or world Cartesian coordinate system. The coordinate system has an origin, which is called the **world origin** and the coordinates of any point defined in that space, are defined with respect to that origin (the point whose coordinates are [0,0,0]). Points are defined in world space (figure 4).

4x4 Matrix Visualized as a Cartesian Coordinate System

Now, let's consider the camera but before we do so, let's talk about a concept in CG which is very important. As you know, objects in 3D can be transformed using any of the three following operators: translation, rotation and scale. If you remember what we said in the lesson dedicated to Geometry, linear transformations (in other words any combination of any of these three operators) can be represented by a 4x4 matrix. If you are not sure why and how this works, read the lesson on Geometry again and particularly the following two chapters: How Does Matrix Work Part 1 & Part 2. Remember that the first three coefficients along the diagonal encode the scale (the coefficients c00, c11 and c22 in the

matrix below), the first three values of the last row encode the translation (the coefficients c30, c31 and c32 — assuming you use the row-major order convention) and the 3x3 upper-left inner matrix encodes the rotation (the red, green and blue coefficients).

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{matrix} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{matrix} \begin{matrix} x-axis \\ y-axis \\ z-axis \\ translation \end{matrix}$$

It might be difficult when you look at the coefficients of a matrix (the actual numbers) to know exactly what the scaling or rotation values are because rotation and scale are sort of combined within the first three coefficients along the diagonal of the matrix. So let's ignore scale for now, and only focus on rotation and translation. As you can see we have nine coefficients that represent a rotation. But how can we interpret what these nine coefficients are? So far, we looked at matrices. But let's now consider what coordinate systems are and by connecting the two together - matrices and coordinate systems - we will answer this question.

The only Cartesian coordinate system we talked about so far, is the **world coordinate system**. This coordinate system is a convention used to define the coordinates [0,0,0] in our 3D virtual space and three unit axes orthogonal to each other (figure 4). It's the prime meridian of a 3D scene, a reference to which any other point or any other arbitrary coordinate system is measured to. Once this coordinate system is defined, we can create other Cartesian coordinate systems and as with points, these coordinates systems are defined by a position in space (a translation value) but also by three unit axes or vectors orthogonal to each other (which by definition what Cartesian coordinate systems are). **Both the position and the values of these three unit vectors are defined with respect to the world coordinate system** as depicted in figure 4.

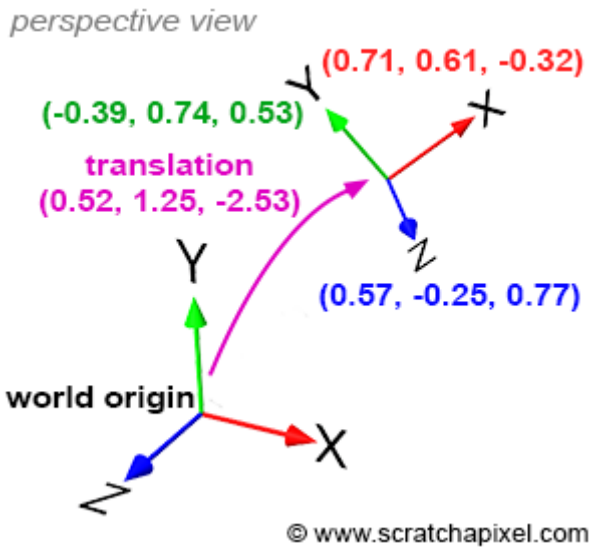


Figure 4: coordinates systems translation and axes coordinates are defined with respect to the world coordinate system (a right-handed coordinate system is used).

The coordinates from figure 4, are, in purple, the position, and in red, green and blue, the coordinates of the x- y- and z-axis of an arbitrary coordinate system (which are all defined with respect to the world coordinate system). Note that the axes making up this arbitrary coordinates system are unit vectors.

The upper-left 3x3 matrix of our 4x4 matrix is actually nothing else than the coordinates of our arbitrary coordinate system's axes. We have three axes, each with three

coordinates which makes nine coefficients. If the 4x4 matrix stores its coefficients using the row-major order convention (this is the convention used by Scratchapixel), then:

- the first three coefficients of the matrix first row (c00, c01, c02), correspond to the coordinates of the coordinate system x-axis,
- the first three coefficients of the matrix second row (c10, c11, c12), are the coordinates of the coordinate system y-axis,
- the first three coefficients of the matrix third row (c20, c21, c22), are the coordinates of the coordinate system z-axis,
- the first three coefficients of the matrix fourth row (c30, c31, c32), are the coordinates of the coordinate system position (translation values).

Here is for example the transformation matrix corresponding to the coordinate system of figure 4:

$$\begin{bmatrix} +0.718762 & +0.615033 & -0.324214 & 0 \\ -0.393732 & +0.744416 & +0.539277 & 0 \\ +0.573024 & -0.259959 & +0.777216 & 0 \\ +0.526967 & +1.254234 & -2.532150 & 1 \end{bmatrix} \begin{matrix} \rightarrow & x-axis \\ \rightarrow & y-axis \\ \rightarrow & z-axis \\ \rightarrow & translation \end{matrix}$$

> In conclusion, we can say that **a 4x4 matrix actually represents a coordinate system** (or reciprocally that any Cartesian coordinate system can be represented by a 4x4 matrix). It's really important that you always see a 4x4 matrix as nothing else than a coordinate system and vice versa (we also sometimes speak of a "local" coordinate system in reference to the "global" coordinate system which in our case, is the world coordinate system).

Local vs. Global Coordinate System

Now that we have established how a 4x4 matrix can be interpreted (and introduced the concept of local coordinate system), let's recall what local coordinate systems are used for. By default, the coordinates of a 3D point are defined with respect to the world coordinate system. The world coordinate system though, is just one among an infinity of possible coordinate systems. But we need a coordinate system to measure all things against by default, so we created one and gave it the special name of "world coordinate system" (it is a convention, like the Greenwich meridian, the meridian at which longitude is defined to be 0). Having one reference is good but not always the best way of keeping track of where

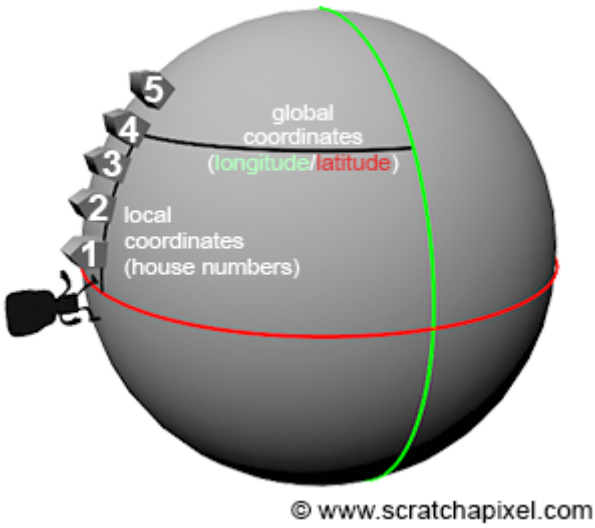


Figure 5: a global coordinate system such as longitude/latitude coordinates can be used to locate a house. We can also locate a house using a numbering system in which the first house defines the origin of a local coordinate system. Note that the

things are in space. For instance, imagine you are looking for a house on a street. If you know the longitude and latitude coordinates of that house, you can always use a GPS to find it. However, if you already are on the street where the house is situated, getting to this house using its number is simpler and quicker than using a GPS. A house number is nothing else than a coordinate defined with respect to a reference, the first house on the street. In this example the street numbers can be seen as a **local coordinate system** (while they can be defined with respect to a global coordinate system, they have their own coordinates with respect to a local reference, the first house on the street), while the longitude/latitude coordinate system can be seen as a **global coordinate system**. Local coordinate system are useful to "find" things when you put "yourself" within the frame of reference in which these things are defined (for example when you are on the street itself). Note that the local coordinate system can itself be defined with respect to the global coordinate system (for instance we can define its origin in terms of latitude/longitude coordinates).

local coordinate system "coordinate" can also be defined with respect to the global coordinate system (i.e. in terms of longitude/latitude coordinates).

Things are the same in CG. It's always possible to know where things are with respect to the world coordinate system, but to simplify calculations, it is often convenient to define things with respect to a local coordinate system (we will show this with an example further down). This what "local" coordinate systems are used for.

When you move a 3D object in a scene such as a 3D cube for instance (but this is true regardless of the object shape or complexity), transformations applied to that object (translation, scale, rotation) can be represented by what we call a 4x4 transformation matrix (it is nothing more than a 4x4 matrix but since it's used to change the position, scale and rotation of that object in space, we call it a transformation matrix). This 4x4 transformation matrix can be seen as the object local frame of reference or local coordinate system. In a way, you don't really transform the object, but transform the local coordinate system of that object, but since the vertices making up the object are defined with respect to that local coordinate system, moving the coordinate system moves the object's vertices with it (see figure 6). **It's important to understand that we don't explicitly transform that coordinate system. We translate, scale and rotate the object, these transformation are represented by a 4x4 matrix, and this matrix can be *visualised* as a coordinate system.**

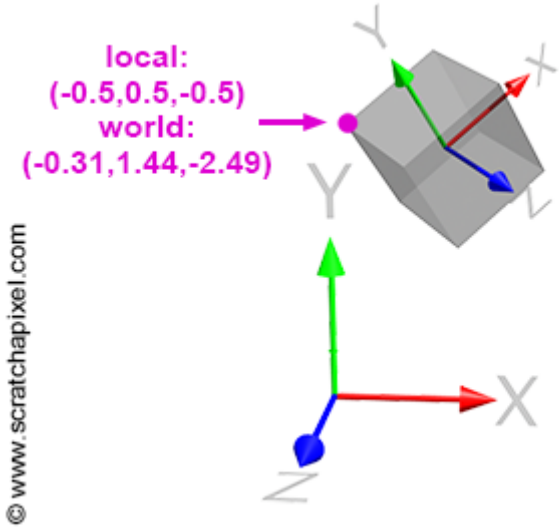


Figure 6: coordinates of a vertex defined with respect to the object local coordinate system and to the world coordinate system.

Transforming Points from One Coordinate System to Another

Note that even though the house is the same, the coordinates of the house depending on whether you use its address or it's longitude/latitude coordinates, are different (as they

relate to the frame of reference in which the location of the house is defined). Look at the highlighted vertex in figure 6. The coordinates of this vertex in the local coordinate system are [-0.5,0.5,-0.5]. But in "world space" (when the coordinates are defined with respect to the world coordinate system), the coordinates are [-0.31,1.44,-2.49]. **Different coordinates, same point**.

As suggested before, it is sometimes more convenient to operate on points when they are defined with respect to a local coordinate system rather than defined with respect to the world coordinate system. For instance, in the example of the cube (figure 6), defining the corners of the cube in local space is easier than in world space. But how do we convert a point or vertex from one coordinate system (such as the world coordinate space) to another (converting points from a coordinate system to another is a very common process in CG)? That's easy. If we know the 4x4 matrix M that transforms a coordinate system A into a coordinate system B, then if we transform a point whose coordinates are originally defined with respect to B with the **inverse of M** (we will explain next why we use the inverse of M rather than M), we get the coordinates of P with respect to A. Let's try with a real example. The matrix M in figure 6 transforming the local coordinate system to which the cube is attached to, is:

$$\begin{bmatrix} +0.718762 & +0.615033 & -0.324214 & 0 \\ -0.393732 & +0.744416 & +0.539277 & 0 \\ +0.573024 & -0.259959 & +0.777216 & 0 \\ +0.526967 & +1.254234 & -2.532150 & 1 \end{bmatrix}$$

By default, the local coordinate system coincides with the world coordinate system (the cube vertices are defined with respect to this local coordinate system). This is illustrated in figure 7a. Then, we apply the matrix M to the local coordinate system, which has for effect to change its position, scale and rotation (this obviously depends on the matrix values). This is illustrated in figure 7b. So before we apply the transform, the coordinates of the highlighted vertex in figure 6 and 7 (the

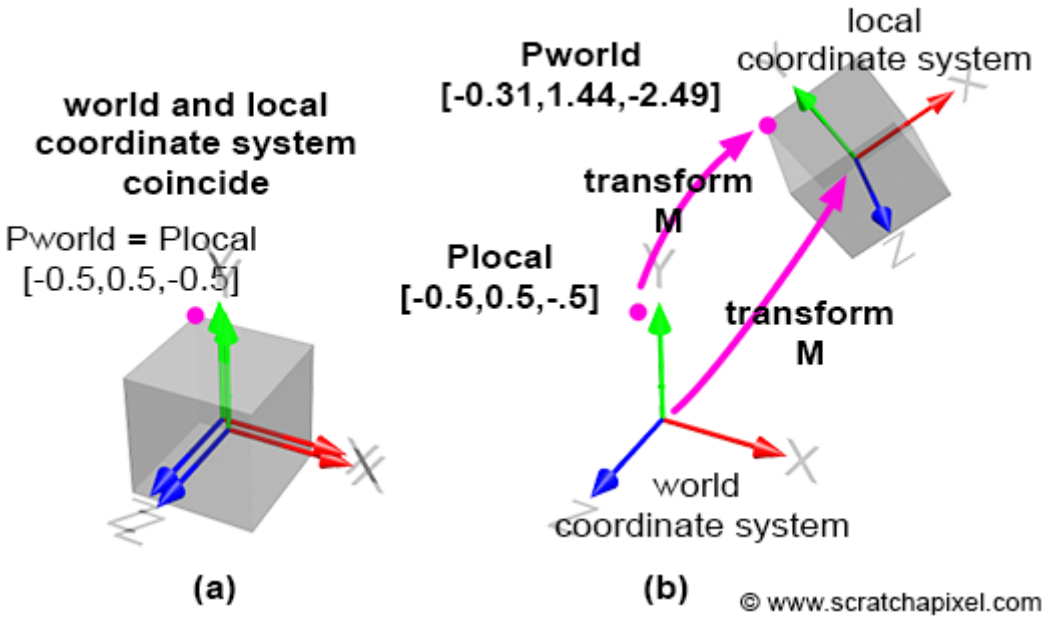


Figure 7: to transform a point which is defined in the local coordinate system to world space, we multiply the point local coordinates by M (in 7a the coordinate systems coincide. They have been shifted slightly to make them visible).

purple dot) are the same in both coordinates systems (since the frames of reference coincide). But after transformation, the world and local coordinates of the points are different (figure 7a and 7b). To calculate the world coordinates of that vertex we need to multiply the point original coordinates by the **local-to-world matrix**: we call it **local-to-world** because it defines the coordinate system with respect to the world coordinate system. This is pretty logical! If you transform the local coordinate system and if you want the cube to move with this coordinate system, you need to apply to the cube vertices the same transform than the one applied to the local coordinate system:

$$P_{world} = P_{local} * M.$$

Obviously, if you now want to go the other way around (to get the point "local coordinates" from its "world coordinates") you need to transform the point world coordinates with the inverse of M:

$P_{local} = P_{world} * M_{inverse}$ or in mathematical notation, $P_{local} = P_{world} * M^{-1}$.
The inverse of M is also called the **world-to-local** coordinate system (it defines where the world coordinate system is with respect to the local coordinate system frame of reference):

$$P_{world} = P_{local} * M_{local-to-world}$$
$$P_{local} = P_{world} * M_{world-to-local}.$$

Let's check that it actually works. Coordinates of the highlighted vertex in local space are [-0.5,0.5,0.5] and in world space: [-0.31,1.44,-2.49]. We also know the matrix M (local-to-world). If we apply this matrix to the point local coordinates, we should obtain the point's world coordinates:

$$P_{world} = P_{local} * M$$
$$P_{world}.x = P_{local}.x * M_{00} + P_{local}.y * M_{10} + P_{local}.z * M_{20} + M_{30}$$
$$P_{world}.y = P_{local}.x * M_{01} + P_{local}.y * M_{11} + P_{local}.z * M_{22} + M_{31}$$
$$P_{world}.z = P_{local}.x * M_{02} + P_{local}.y * M_{12} + P_{local}.z * M_{22} + M_{32}$$

Let's implement and check the results (you can use the code from the Geometry lesson):

```
001 Matrix44f m(0.718762, 0.615033, -0.324214, 0, -0.393732, 0.744416, 0.539277, 0,
002 Vec3f Plocal(-0.5, 0.5, -0.5), Pworld;
003 m.multVecMatrix(Plocal, Pworld);
004 std::cerr << Pworld << std::endl;
```

The output is:

```
001 (-0.315792 1.4489 -2.48901)
```

Let's now transform the world coordinates of this point, to local coordinates. Our implementation of the Matrix class contains a method to invert the current matrix. We will use it to compute the world-to-local transformation matrix, and then apply this matrix to the point world coordinates:

```
Matrix44f m(0.718762, 0.615033, -0.324214, 0, -0.393732, 0.744416, 0.539277, 0,
m.invert();
Vec3f Pworld(-0.315792, 1.4489, -2.48901), Plocal;
m.multVecMatrix(Pworld, Plocal);
std::cerr << Plocal << std::endl;
```

The output is:

```
(-0.500004 0.499998 -0.499997)
```

The coordinates are not exactly (-0.5, 0.5, -0.5) because of some floating point precision issue and also because we've truncated the input point world coordinates but if we round it off to one decimal place, we get (-0.5, 0.5, -0.5) which is the correct result.

> At this point of the chapter, you should understand the difference between the world/global and local coordinate system, and how to transform points or vectors from one system to the other (and vice versa).
>
> When we transform a point from the world to the local coordinate system (or the other way around), we often say that we go from **world space** to **local space**. We will use this terminology often.

Camera Coordinate System and Camera Space

A camera, in CG (as well as in the real world) is no different than any other 3D object. When you take a photograph, you need to move and rotate the camera to adjust the viewpoint. So in a way, when you transform a camera (by translating and rotating it — note that scaling a camera doesn't make much sense), what you actually do is **transforming a local coordinate system which implicitly represents the transformations applied to that camera**. In CG we call this spatial reference system (the term spatial reference system or reference, is sometimes used in place of coordinate system), the **camera coordinate system** (you might also find it called the eye coordinate system in other references). This is an important coordinate system and we will explain why in a moment.

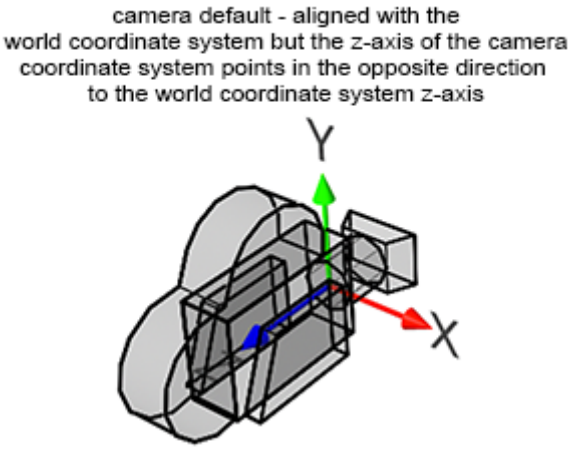


Figure 8: when you create a camera, it is by default aligned along the world coordinate system negative z-axis. This is a convention used by most 3D applications.

So a camera really is nothing else that a coordinate system and thus the technique to transform points from one coordinate system

to another we described just before, can also be applied here to transform points from the world coordinate system to the camera coordinate system (and vice versa). We say that we transform points from **world space to camera space** (or camera space to world space if we apply the transformation the other way around).

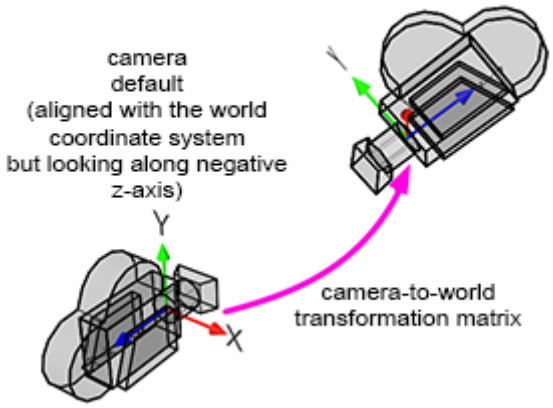


Figure 9: transforming the camera coordinate system with the camera-to-word transformation matrix.

Note though that cameras always point along the world coordinate system negative z-axis. If you look at figure 8, you will see that the camera z-axis is actually pointing in the opposite direction to the world coordinate system z-axis (when the x-axis points to the right, the z-axis goes inward, into the screen, rather than outward).

> Cameras point along the world coordinate system negative z-axis so that when a point is converted from world space to camera space (and then later from camera space to screen space), if the point is to left of the world coordinate system y-axis, it will also map to the left of the camera coordinate system y-axis. In other words, we need the x-axis of the camera coordinate system to point to the right when the world coordinate system x-axis also points to the right; and the only way you can get that configuration, is by having camera looking down the negative z-axis.

Because of this, the sign of the z coordinate of points is inverted when we go from one system to the other. Keep this in mind, as it will play a role when we will (finally) get to studying the perspective projection matrix.

To summarize: if we want to convert the coordinates of a point in 3D from world space (which is the space in which points are defined in a 3D scene), to the space of a local coordinate system, we need to multiply the point world coordinates by the inverse of the local-to-world matrix.

Of the Importance of Converting Points to Camera Space

This a lot a reading but what for? Because, as already suggested we are now going to show that to "project" a point on the canvas (the 2D surface on which we will draw an image of the 3D scene), we will need to convert or transform points from world to camera space. And here is why.

Let's recall that what we are trying to achieve, is to compute P', the coordinates of a point P from the 3D scene on the surface of a canvas, which as mentioned before, is the 2D surface on which the image of the scene will be drawn (the canvas is

also called the projection plane or in CG, the **image plane**). If you trace a line from P to the eye (the origin of the camera coordinate system), P' is the line's point of intersection with the canvas (figure 10). When the point P coordinates are defined with respect to the camera coordinate system, computing the position of P' is trivial. If you look at figure 10 representing a side view of our setup, you can see that by construction we can trace two triangles $\triangle ABC$ and $\triangle AB'C'$, where A is the eye, B is the distance from the eye to P along the camera coordinate system z-axis, C is the distance from the eye to P along the camera coordinate system y-axis, B' is the distance from the eye to the canvas (for now, we will assume that this distance is 1 which is going to simplify our calculations) and C' is the distance from the eye to the P' along the camera coordinate system y-axis. The triangles $\triangle ABC$ and $\triangle AB'C'$ are said to be similar (similar strangles have the same shape, but eventually different sizes). Similar triangles have an interesting property: the ratio between their adjacent and opposite sides is the same. In other words:

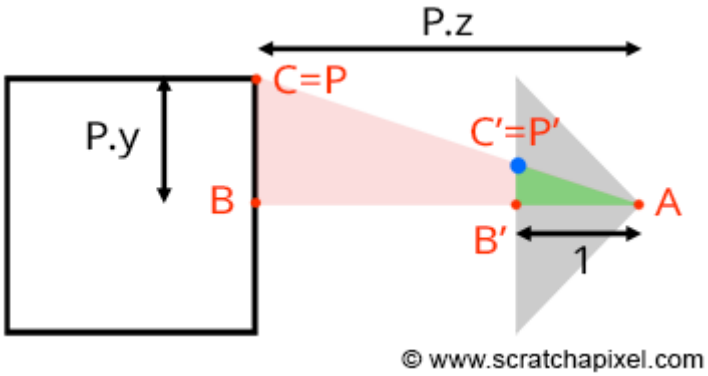


Figure 10: the coordinates of the point P', the projection of P on the canvas can be computed using simple geometry. The rectangle ABC and AB'C' are said to be similar (side view).

$$\frac{BC}{AB} = \frac{B'C'}{AB'}.$$

Because the canvas is 1 unit away from the origin, we know that AB' equals 1. We also know the position of B and C which are the point P z- (depth) and y-coordinate (height) (assuming P's coordinates are defined with respect to the camera coordinate system). If we substitute these numbers in the above equation, we get:

$$\frac{P.y}{P.z} = \frac{P'.y}{1}.$$

Where y' is the y coordinate of P'. Thus:

$$P'.y = \frac{P.y}{P.z}.$$

This is probably one the simplest and most fundamental relation in computer graphics, known as the z or **perspective divide**. The exact same principle applies to the x coordinate. The projected point x coordinate (x') is the corner's x coordinate divided by its z coordinate:

$$P'.x = \frac{P.x}{P.z}.$$

> We described this method several times in the different lessons of the website, but what we want to show here, is that **in order to compute P' using these equations,**

> **the coordinates of P ought to be defined with respect to the camera coordinate system. However, points from the 3D scene are originally defined with respect to the world coordinate system. Therefore, the first and foremost operation we need to apply to points before projecting them onto the canvas, is to convert them from world space to camera space**.

How to do we do that? If we know the camera-to-world matrix (which is similar to the local-to-camera matrix we studied in the previous case), we can transform any point which coordinates are defined in world space to camera space, by multiplying this point by the camera-to-world inverse matrix (the world-to-camera matrix):

$$P_{camera} = P_{world} * M_{world-to-camera}.$$

Then at this point we can "project" the point on the canvas using the equations we presented before:

$$P'.x = \frac{P_{camera}.x}{P_{camera}.z}$$
$$P'.y = \frac{P_{camera}.y}{P_{camera}.z}.$$

Recall that by default, cameras are oriented along the world coordinate system's negative z-axis. This means that when we convert a point from world space to camera space, the sign of the point z-coordinate in camera space is necessarily reversed; it becomes negative if the z-coordinate was positive in world space, and positive after the conversion if it was originally negative. **Note that a point defined in camera space can only be visible, if its z-coordinate is negative** (take a moment to verify this statement). As a result, when the x- and y-coordinate of the original point are divided by the point's negative z-coordinate, the sign of the resulting projected point's x and y-coordinates is reversed as well. This is a problem because a point which is situated to the right of the screen coordinate system y-axis when you look through the camera or a point which appears above the horizontal line passing through the middle of the frame, ends up either to the left of the vertical line or below the horizontal line once projected. The point's coordinates are mirrored. The solution to this problem is simple. We just need to make the point z-coordinate positive which we can easily do by reversing its sign at the time that the projected point's coordinates are computed:

$$P'.x = \frac{P_{camera}.x}{-P_{camera}.z}$$
$$P'.y = \frac{P_{camera}.y}{-P_{camera}.z}.$$

To summarize: points in a scene are defined in the world coordinate space. However to project them onto the surface of the canvas, we first need to convert the 3D point coordinates from world space to camera space. This can be done by multiplying the point world coordinates by the inverse of the camera-to-wold matrix. Here is the code for performing this conversion:

```
Matrix44f cameraToWorld(0.718762, 0.615033, -0.324214, 0, -0.393732, 0.744416,
Matrix4ff worldToCamera = cameraToWorld.inverse();
Vec3f Pworld(-0.315792, 1.4489, -2.48901), Pcamera;
worldToCamera.multVecMatrix(Pworld, Pcamera);
std::cerr << Pcamera << std::endl;
```

We can use the resulting point now in camera space, to compute its 2D coordinates on the canvas, using the perspective projection equations (dividing the point coordinates with the **inverse** of the point's z-coordinate).

From Screen Space to Raster Space

At this point, we know how to compute the projection of a point on the canvas. We first need to transform points from world space to camera space, and then divide the points x and y coordinates by their respective z-coordinate. Let's recall that the canvas lies on what we call in CG, the **image plane**. So you now have a point P' lying on the image plane, which is the projection of P onto that plane. **But in which space are the coordinates of P' defined into?** Note that because the point P' lies on a plane, we are not interested in P' z-coordinate anymore. In other words, we don't need to declare P' as a 3D point; a 2D point suffices (this is in fact partially true. To solve the visibility problem, the rasterization algorithm uses the z-coordinates of the projected points. However we will ignore this technical detail for now).

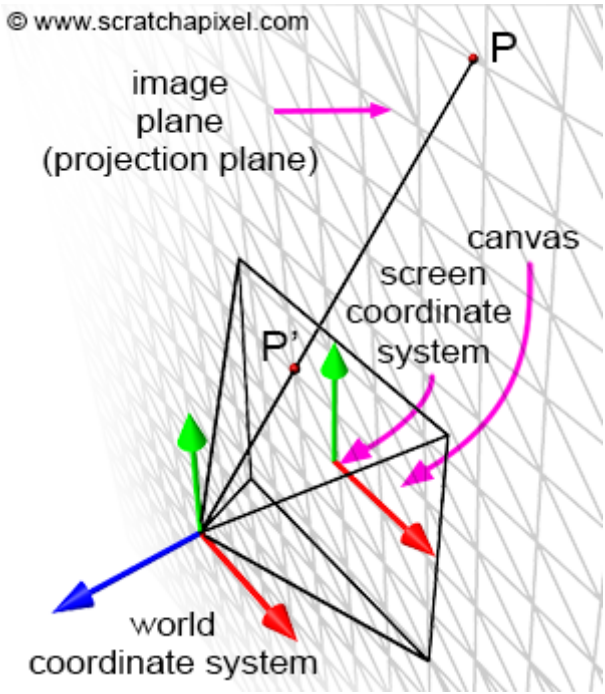
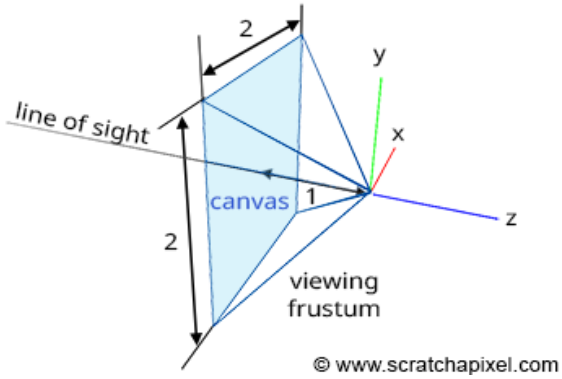


Figure 11: the screen coordinate system is a 2D Cartesian coordinate system. It marks the centre of the canvas. The image plane is infinite, but the canvas actually delimits the surface over which the image of the scene will be drawn onto. The canvas size can have any size. In this example, it is two units long in both dimension (as with every Cartesian coordinate system, the screen coordinate system's axes have unit length).

Since P' is a 2D point, it is defined with respect to a 2D coordinate system which in CG we call the **image or screen coordinate system**. This coordinate system marks the centre of the canvas; the coordinates of any point projected onto the image plane refer to this coordinative system. 3D points with positive x-coordinates are projected to the right of the image coordinate system y-axis. 3D points with positive y-coordinates are projected above the image coordinate system x-axis (figure 11). The image plane is a plane so technically it is



infinite. But images are not infinite in size; they have a width and a height. Thus, we will cut off a rectangular shape centred around the image coordinate system which we will define as the "bounded region" over which the image of the 3D scene will be drawn (figure 11). You can see this region as the paintable or drawable surface of a canvas. The dimension of this rectangular region can be anything we want. Changing its size changes the extent of a given scene imaged by the camera (figure 13). We will study the effect of the canvas size in the next lesson. In figure 12 and 14 (top), the canvas is 2 units long in each dimension (vertical and horizontal).

Any projected point whose absolute x and y coordinate is greater than half of the canvas' width **or** half of the canvas' height respectively is not visible in the image (is clipped).

Figure 12: in this example, the canvas is 2 units along the x-axis and 2 units along the y-axis. You can change the dimension of the canvas if you wish. By making it bigger or smaller you will see more or less of the scene.

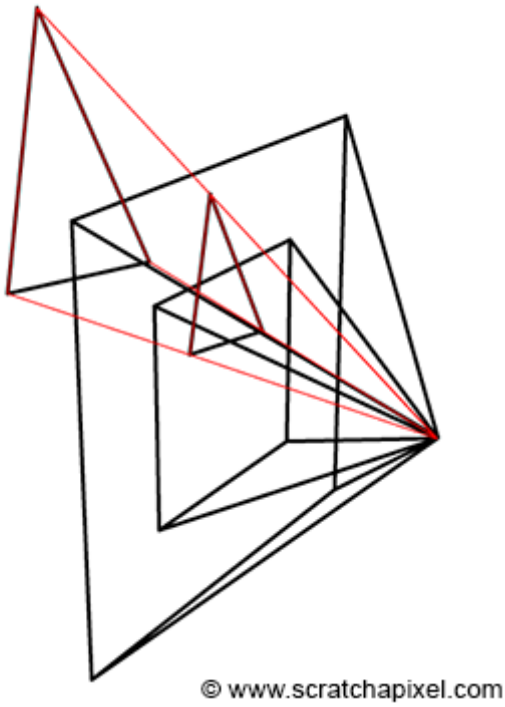


Figure 13: changing the dimension of the canvas changes the extent of a given scene that is imaged by the camera. In this particular example, two canvases are represented. On the smaller one, the triangle is only partially visible. On the larger one, the entire triangle is visible. Canvas size and field-of-view relate to each other.

$$\text{visible} = \begin{cases} yes & |P'.x| \leq \frac{W}{2} \text{ or } |P'.y| \leq \frac{H}{2} \\ no & \text{otherwise} \end{cases}$$

|a| means in mathematics, the absolute value of a. The variables W and H are the width and the height of the canvas.

If the coordinates of P are real numbers (floats or doubles in programming), P' coordinates are also real numbers. If P' coordinates are within the canvas boundaries, then P' is visible, otherwise the point is not visible and we can ignore it. If P' is visible it should appear as a dot in the image. A dot in a digital image is a pixel. Note that pixels too are 2D points, only their coordinates are integers and the coordinate system these coordinates refer to is located in the upper-left corner of the

image. Its x-axis points to the right (when the world coordinate system x-axis points to the right), and its y-axis points downwards (figure 14). This coordinate system in CG, is called the **raster coordinate system**. A pixel in this coordinate system, is one unit long in x and y. What we need to do, is convert P' coordinates which are defined with respect to the image or screen coordinate system into pixel coordinates (what's the position of P' in the image in terms of pixel coordinates). This is again, another change of coordinate system; we say that we need to go from **screen space to raster space**. How do we do that?

The first thing we are going to do is to remap P' coordinates in the range [0,1]. This is mathematically easy. Since we know the dimension of the canvas, all we need to do is apply the following formulas:

$$P'_{normalized}.x = \frac{P'.x + width/2}{width}$$
$$P'_{normalised}.y = \frac{P'.y + height/2}{height}$$

Because the coordinates of the projected point P' are now in the range [0,1], we say that the coordinates are normalized. For this reason, we also call the coordinate system in which the points are defined after normalization, the **NDC coordinate system** or **NDC space**. NDC stands for **Normalized Device Coordinate**. The NDC coordinate system's origin is situated in the lower-left corner of the canvas. Note that at this point, the coordinates are still real numbers, only they are now in the range [0,1].

The last step is simple. All we need to do is now multiply the projected point x- and y-coordinates in NDC space, by the actual image pixel width and pixel height respectively. This is a simple remapping of the range [0,1] to the range [0,Pixel Width] for the x-coordinate, and [0,Pixel Height] for the y-coordinate

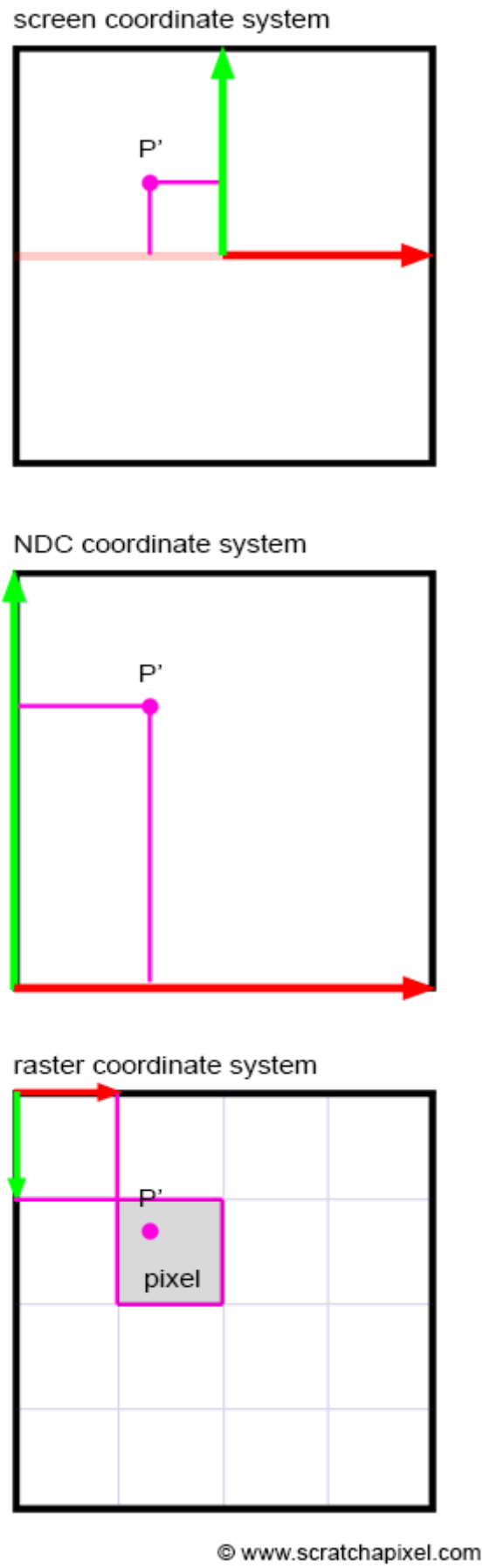


Figure 14: to convert P' from screen space to raster space, we first need to go from screen space (top) to NDC space (middle), then NDC space to raster space (bottom). Note that the y-axis of the NDC coordinate system goes up, but that the y-axis of the raster coordinate system goes down. This implies that we

invert P' y-coordinate when we go from NDC to raster space.

respectively. Now though that pixel coordinates are integers, thus rounding off the resulting numbers to the smallest following integer value is necessary (to do that, we will use the mathematical floor function; it rounds off a real number to its smallest following integer). After this final step, P' coordinates are defined in **raster space**:

$$P'_{raster}.x = \lfloor P'_{normalized}.x * \text{ Pixel Width} \rfloor$$
$$P'_{raster}.y = \lfloor P'_{normalized}.y * \text{Pixel Height} \rfloor$$

In mathematics, $\lfloor a \rfloor$, denotes the floor function. Pixel width and pixel height are the actual dimension of the image in pixels. There is a small detail though we need to take care of. The y-axis in the NDC coordinate system points up while in the raster coordinate system points down. Thus, to go from one coordinate system to the other, P' y-coordinate also needs to be inverted. We can easily account for this by doing a small modification to the above equations:

$$P'_{raster}.x = \lfloor P'_{normalized}.x * \text{ Pixel Width} \rfloor$$
$$P'_{raster}.y = \lfloor (1 - P'_{normalized}.y) * \text{Pixel Height} \rfloor$$

In OpenGL, the conversion from NDC space to raster space, is called the viewport transform. In fact what we call the canvas in this lesson is generally called in CG the **viewport**. The viewport though means different things to different people. To some it designates the "normalised window" of the NDC space. To others it actually designates the window in pixels on the screen, in which the final image is displayed.

Done! You have converted a point P defined in world space into a visible point in the image, which pixel coordinates you have computed using a series of operations converting this point from world space to camera space, camera space to screen space, screen space to NDC space and finally, NDC space to raster space.

Summary

Because this process is so important and fundamental we will summarize everything we've learned in this chapter.

- Points in a 3D scene are defined with respect to the world coordinate system.
- A 4x4 matrix can be seen as a "local" coordinate system.
- We learned how to convert points from world to any local coordinate system. If we know the local-to-world matrix, we can multiply the world coordinate of the point by the inverse of the local-to-world matrix (the world-to-local matrix).
- We also use 4x4 matrices to transform cameras. Therefore, we can also convert points from world space to camera space.
- Computing the coordinates of a point from camera space onto the canvas, can be done using perspective projection. It requires a simple division of the point's x- and

y-coordinate by the point's z-coordinate. Before projecting the point onto the canvas, we need to convert the point from world space to camera space. The resulting projected point is defined in image space, and is a 2D point (the z-coordinate can be discarded).

- We then to convert the 2D point in image space to NDC space. In NDC space, the coordinates of the point are remapped to the range [0,1].
- Finally, we convert the 2D point in NDC space to raster space. For doing so we just need to multiply the points NDC x and y coordinates by the image width and height in pixels. Pixel coordinates are integers rather than real numbers, thus they need to be rounded off to the smallest following integer when converted from NDC space to raster space. Because in raster space, the y-axis is located in the upper-left corner of the image and is pointing down while the NDC coordinate system is located in the lower-left corner of the image with its y-axis pointing up, the y-coordinates needs to be inverted in the process.

Space	Description
World Space	The space in which the points are originally defined in the 3D scene. Coordinates of point in world space are defined with respect to the world Cartesian coordinate system.
Camera Space	The space in which points are defined with respect to the camera coordinate system. To convert points from world to camera space, we need to multiply points in world space by the inverse of the camera-to-world matrix. By default the camera is located at the origin and is oriented along the world coordinate system negative z-axis. Once points are in camera space, they can be projected on the canvas using perspective projection.
Screen Space	In this space, points are in 2D. They lie in the image plane. Because the plane is infinite, the canvas defines the region of this plane on which the scene can be drawn. The size of the canvas is arbitrary and defines "how much" of the scene we see. The image or screen coordinate system marks the centre of the canvas (and the centre of the image plane). If a point on the image plane is outside the boundaries of the canvas, it is not visible. It is of course visible otherwise.
NDC Space	2D point lying in the image plane and contained within the boundaries of the canvas are then converted to NDC space. The principle is to normalize the point's coordinates, in other words, to remap them to the range [0,1]. Note that NDC coordinates are still real numbers.
Raster Space	Finally, 2D points in NDC space are converted to 2D pixel coordinates. For doing so we multiply the normalized points x and y coordinates by the image width and height in pixels. Going from NDC to raster space also requires the y-coordinate of the point to be inverted. Final coordinates need

to be rounded off to the nearest following integers (pixel coordinates are integers).

Code

The following function converts a point from 3D world coordinates to 2D pixel coordinates. This implementation is quite naive but we didn't write it for efficiency. We wrote it so that every step is clearly visible and contained within a single function.

```
001 bool computePixelCoordinates(
002     const Vec3f &pWorld,
003     const Matrix44f &cameraToWorld,
004     const float &canvasWidth,
005     const float &canvasHeight,
006     const int &imageWidth,
007     const int &imageHeight,
008     Vec2i &pRaster)
009 {
010     // First transform the 3D point from world space to camera space.
011     // It is of course inefficient to compute the inverse of the cameraToWorld
012     // matrix in this function. It should be done outside the function, only on
013     // and the worldToCamera should be passed to the function instead.
014     // We are only compute the inverse of this matrix in this function ...
015      Vec3f pCamera;
016      Matrix44f worldToCamera = cameraToWorld.inverse();
017      worldToCamera.multVecMatrix(pWorld, pCamera);
018      // Coordinates of the point on the canvas. Use perspective projection.
019      Vec2f pScreen;
020      pScreen.x = pCamera.x / -pCamera.z;
021      pScreen.y = pCamera.y / -pCamera.z;
022      // If the x- or y-coordinate absolute value is greater than the canvas wid
023      // or height respectively, the point is not visible
024      if (std::abs(pScreen.x) > canvasWidth || std::abs(pScreen.y) > canvasHeigh
025          return false;
026      // Normalize. Coordinates will be in the range [0,1]
027      Vec2f pNDC;
028      pNDC.x = (pScreen.x + canvasWidth / 2) / canvasWidth;
029      pNDC.y = (pScreen.y + canvasHeight / 2) / canvasHeight;
030      // Finally convert to pixel coordinates. Don't forget to invert the y coor
031      pRaster.x = std::floor(pNDC.x * imageWidth);
032      pRaster.y = std::floor((1 - pNDC.y) * imageHeight);
033
034      return true;
035 }
036
037 int main(...)
038 {
039     ...
040     Matrix44f cameraToWorld(...);
041     Vec3f pWorld(...);
042     float canvasWidth = 2, canvasHeight = 2;
043     uint32_t imageWidth = 512, imageHeight = 512;
```

```
044     // The 2D pixel coordinates of pWorld in the image if the point is visible
045     Vec2i pRaster;
046     if (computePixelCoordinates(pWorld, cameraToWorld, canvasWidth, canvasHeigh
047        std::cerr << "Pixel coordinates " << pRaster << std::endl;
048     }
049     else {
050        std::cert << Pworld << " is not visible" << std::endl;
051     }
052     ...
053 
054     return 0;
055 }
```

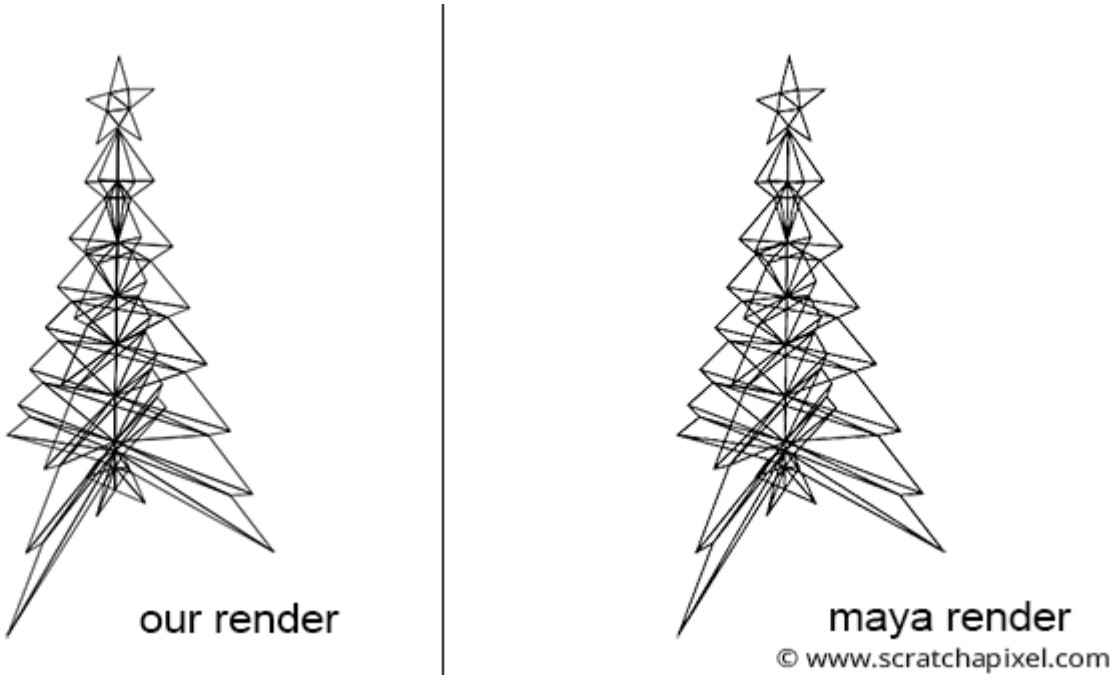
We will use a similar function in our example program (look at the source code chapter). To demonstrate the technique, we created a simple object in Maya (a tree with a star sitting on top) and rendered an image of that tree from a given camera in Maya (see the image below). To simplify the exercise we triangulated the geometry. We then stored a description of that geometry and the Maya camera 4x4 transform matrix (the camera-to-world matrix) in our program. To create an image of that object, we need to loop over each triangle making up the geometry, extract from the vertex list the vertices making up the current triangle, convert these vertices world coordinates to 2D pixel coordinates, and finally draw lines connecting the resulting 2D points to draw an image of that triangle as viewed from the camera (we trace a line from the first point to the second point, from the second point to the third, and then from the third point back to the first point). We will store the resulting lines to a SVG file. The SVG format is designed to create images using simple geometric shapes such as lines, rectangles, circles, etc. which are described in XML. Here is how we define a line in SVG for instance:

```
<line x1="0" y1="0" x2="200" y2="200" style="stroke:rgb(255,0,0);stroke-width:2" />
```

SVG files themselves can be read and displayed as images by most Internet browsers. Storing the result of our programs in SVG is very convenient. Rather than rendering these shapes ourselves, we can simply store their description in a SVG file and have other applications rendering the final image for us (we don't need to care for anything that relates to rendering these shapes and displaying the image to the screen which is far from being obvious from a programming point of view).

The complete source code of this program can be found in the source code chapter. Finally here is the result of our program (left) compared to a render of the same geometry from the same camera in Maya (right). As expected the visual results are exactly the same (you can read the SVG file produced by the program in any Internet browser).

If you wish to reproduce this result in Maya, you will need to import the geometry (which we provide in the next chapter as an obj file, create a camera, set its angle of view to 90 degrees (we will explain why in the next lesson), and make the film gate square (by setting up the vertical and horizontal film gate parameters to 1). Set the render resolution to 512x512 and render from Maya. You then need to export the camera's transformation



matrix using for example the following Mel command:

```
getAttr camera1.worldMatrix;
```

Set the camera-to-world matrix in our program with the result of this command (the 16 coefficients of the matrix). Compile the source code, and run the program. The result exported to the SVG file should match Maya's render.

What Else?

This chapter contains a lot of information. Most resources devoted to the process focus their explanation on the perspective process but forget to mention everything that comes before and after the perspective projection itself such as the world to camera transform, or the conversion of the screen coordinates to raster coordinates. Our goal what for you to be able to produce an actual result at the end of this lesson, which we could also match to a render from a professional 3D application such as Maya. We wanted you to have a complete picture of the process from beginning to end. However dealing with cameras is slightly more complicated than just what we've described in this chapter. For instance, if you used a 3D program before, you are probably familiar with the fact that the camera transform is not the only parameter you can change to adjust what you see in the camera's view. You can also change for example its focal length. How does the focal length affect the result of the conversion process is not something we have explained in this lesson. The near and far clipping planes associated with cameras are also having an effect on the perspective projection process and more notably the perspective and orthographic projection matrix. In this lesson we assumed that the canvas was located one unit away from the camera coordinate system. However, this is not always the case and can be controlled through the near clipping plane. How do we compute pixel coordinates then, when the distance between the camera coordinate system's origin and the canvas is different than 1? All these unanswered questions will be addressed in the next lesson, devoted to the topic of 3D viewing.

Exercises

- Change the canvas dimension in the program (the canvasWidth and canvasHeight parameters). Keep the value of the two parameters equal. What happens when the values get smaller? What happens when they get bigger?

← Previous Chapter Chapter 2 of 3 Next Chapter →