**Chapter 7**

**Three Dimensional Basics[[1]](#footnote-0)**

The three-dimensional transformations are extensions of two-dimensional transformation. In 2D two coordinates are used, i.e., x and y whereas in 3D three co-ordinates x, y, and z are used.

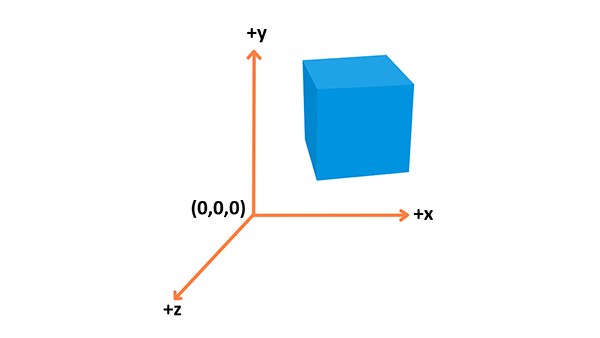
For three dimensional images and objects, three-dimensional transformations are needed. These are translations, scaling, and rotation. These are also called as basic transformations are represented using matrix. More complex transformations are handled using matrix in 3D.

The 2D can show two-dimensional objects. Like the Bar chart, pie chart, graphs. But some more natural objects can be represented using 3D. Using 3D, we can see different shapes of the object in different sections.

In 3D when a translation is done we need three factors for rotation also, it is a component of three rotations. Each can be performed along any three Cartesian axis. In 3D also we can represent a sequence of transformations as a single matrix.

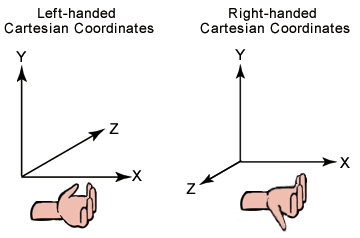
# Coordinate system[[2]](#footnote-1)

3D essentially is all about representations of shapes in a 3D space, with a coordinate system used to calculate their position.



the x axis points to the right, the y axis points up, and the z axis points out of the screen, as seen in the above diagram.

3-D graphics applications use two types of Cartesian coordinate systems: left-handed and right-handed[[3]](#footnote-2). In both coordinate systems, the positive x-axis points to the right, and the positive y-axis points up. You can remember which direction the positive z-axis points by pointing the fingers of either your left or right hand in the positive x direction and curling them into the positive y direction. The direction your thumb points, either toward or away from you, is the direction that the positive z-axis points for that coordinate system. The following illustration shows these two coordinate systems.

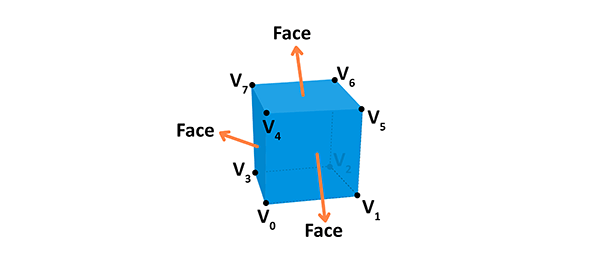


# Objects

Different types of objects are built using vertices. A **Vertex** is a point in space having its own 3D position in the coordinate system and usually some additional information that defines it. Every vertex is described by these attributes:

* **Position**: Identifies it in a 3D space (x, y, z).
* **Color**: Holds an RGBA value (R, G and B for the red, green, and blue channels, alpha for transparency — all values range from 0.0 to 1.0).
* **Normal:** A way to describe the direction the vertex is facing.
* **Texture**: A 2D image that the vertex can use to decorate the surface it is part of instead of a simple color.

You can build geometry using this information — here is an example of a cube:

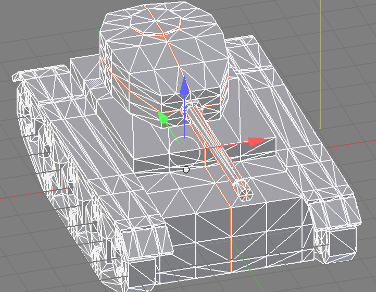


A face of the given shape is a plane between vertices. For example, a cube has 8 different vertices (points in space) and 6 different faces, each constructed out of 4 vertices. A normal defines which way the face is directed in. Also, by connecting the points we're creating the edges of the cube. The geometry is built from a vertex and the face, while material is a texture, which uses a color or an image. If we connect the geometry with the material, we will get a mesh.

## What is a Mesh[[4]](#footnote-3)

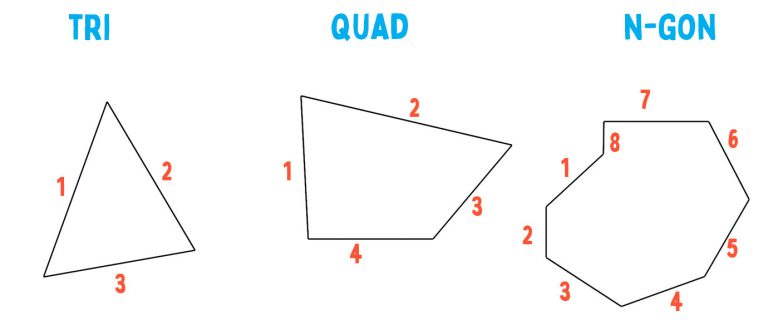
A mesh is a collection of vertices, edges, and faces that define the shape of a 3D object. The most popular type of polygon mesh used in computer graphics is a Triangle Mesh. Any object, either a mountain or a game character, can be modeled with Triangle meshes.

For example, the image below shows triangle meshes making up a tank.



The vertices of a mesh are the inputs to a Rendering Pipeline. These vertices go through several stages, such as coordinate transformation and rasterization, before they end up in the framebuffer, i.e., the screen.

A face can have a minimum of 3 vertices[[5]](#footnote-4). This is then a **triangle**. If you have four vertices it is called a **quad**, and if you have more than four vertices, then it is simply a **general polygon**.



When you have 5 or more edges around a face, it’s referred to as an **N-gon**. They get that name just because it represents a polygon with n-number of sides. N can represent any number, really. So, for instance, an 8 sided face is an N-gon, or 8-gon.[[6]](#footnote-5)

## Normals[[7]](#footnote-6)

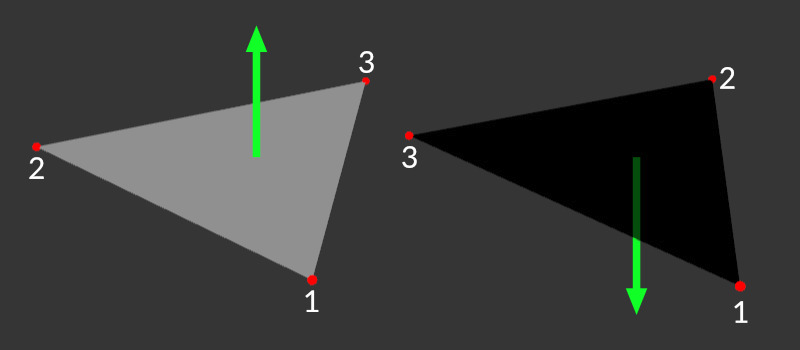
There are two types of normals, Face normals and vertex normals.

### Face normals

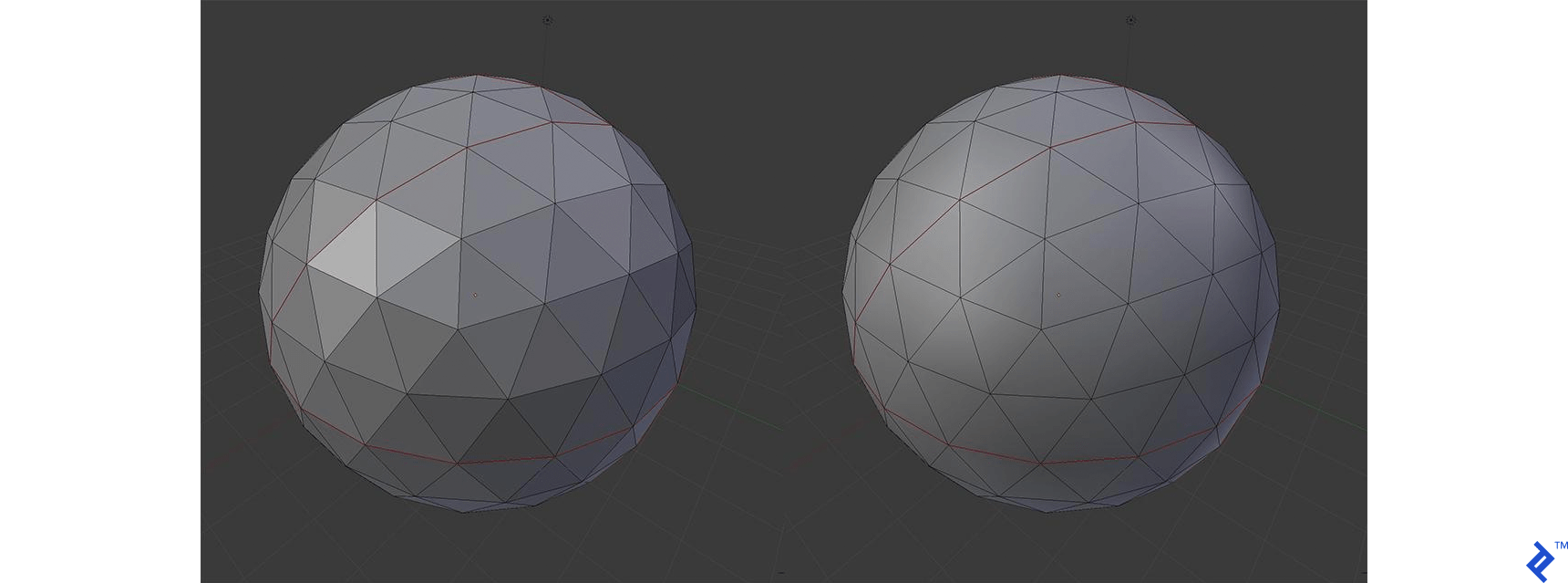
A normal is just a **hidden vector**, basically a direction without a specified origin.

* If the vertex order is **clockwise**, the normal points **upwards**;
* if it is **counter-clockwise** it's pointing **down**.

You may know those faces as backfaces, the "back side" of polygons (note: can vary between 3d applications). The 3d application uses that vertex order, and consequently the face normal, to determine what is the inside and what is the outside of an object. Most game engines do not render the backfaces of your models unless explicitly told to do so.

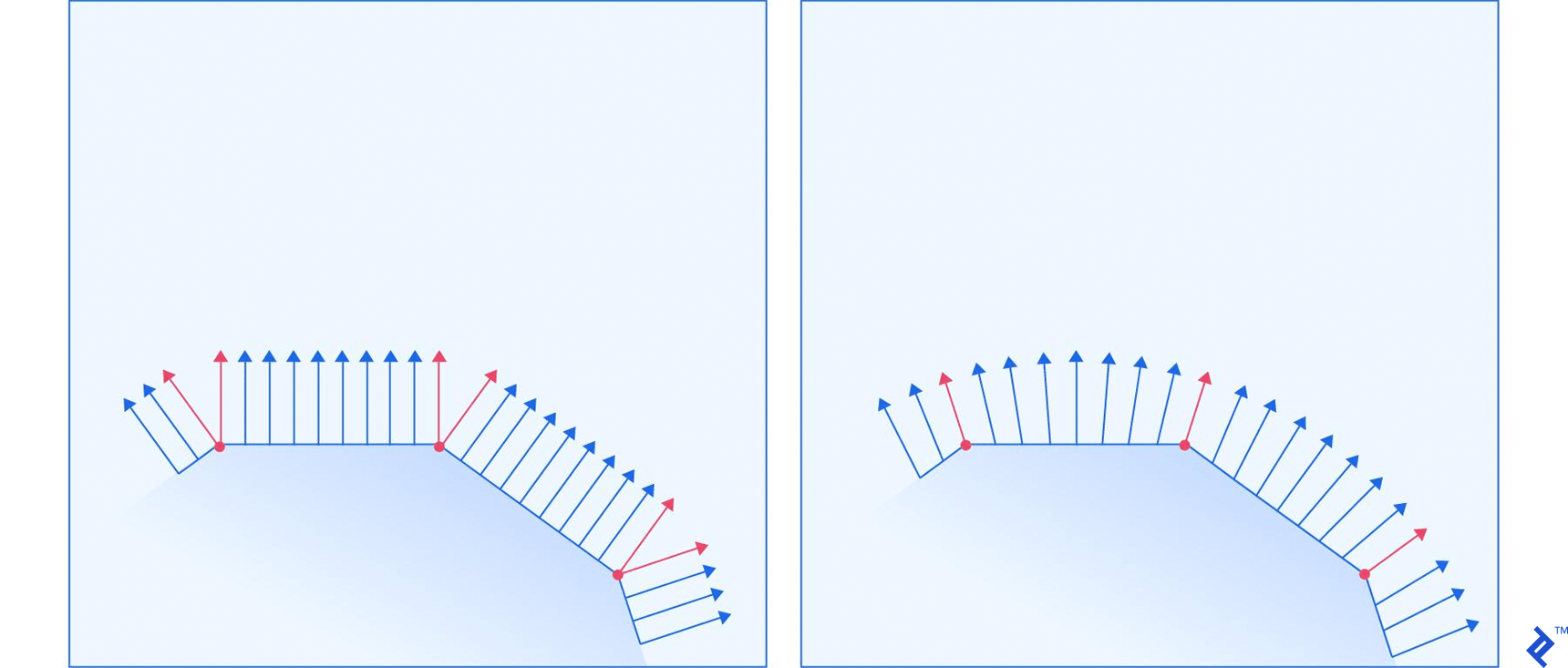


### Vertex Normal[[8]](#footnote-7)



Consider the two models above. They consist of the same vertex positions, yet look totally different when rendered. How is that possible?

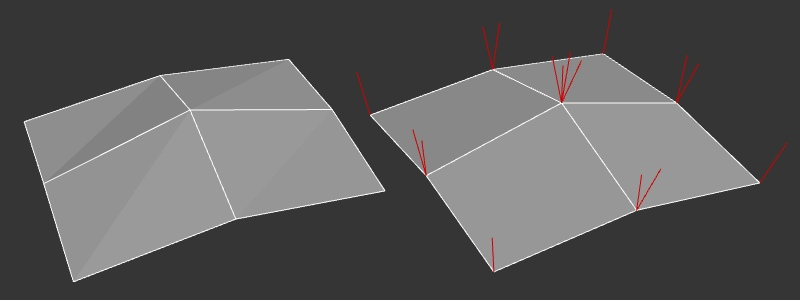
Besides telling the renderer where we want a vertex to be located, we can also give it a hint on how the surface slopes يميل in that exact position. The hint is in the form of the normal of the surface at that specific point on the model, represented with a 3D vector. The following image should give you a more descriptive look at how that is handled.



The left and right surface correspond to the left and right ball in the previous image, respectively. The red arrows represent normals that are specified for a vertex, while the blue arrows represent the renderer’s calculations of how the normal should look for all the points between the vertices. The image shows a demonstration for 2D space, but the same principle applies in 3D.

The normal is a hint for how lights will illuminate the surface. The closer a light ray’s direction is to the normal, the brighter the point is. Having gradual changes in the normal direction causes light gradients, while having abrupt changes with no changes in-between causes surfaces with constant illumination across them, and sudden changes in illumination between them.

Each **vertex normal is a vector pointing in some direction[[9]](#footnote-8)**. In the case of a single triangle, all the vertex normals point in the same direction as the face normal - unless explicitly changed by the artist/designer. This is not particularly useful for a single triangle but it gets interesting if we have neighboring polygons.

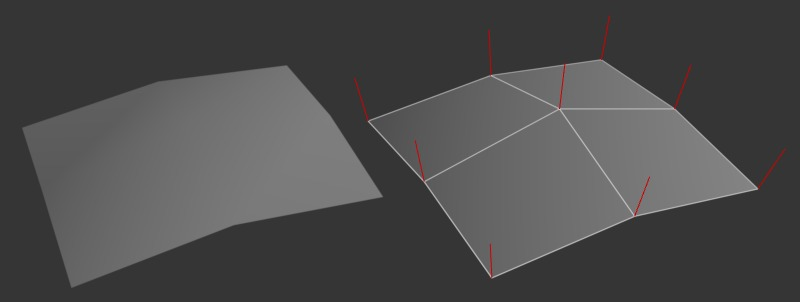


vertex normals in a multi-polygon mesh

Now let's look at a 4x4 quad mesh which is slightly rounded in the middle. On the left is our faceted mesh, we can even see the triangulation. On the right, we can see the individual red vertex normals. Each polygon has 4 vertices and thus 4 vertex normals, despite several polygons sharing the same vertices. Vertices that are shared by multiple faces also have the same number of red normal indicators pointing upwards.

So what can we do with those vertex normals?

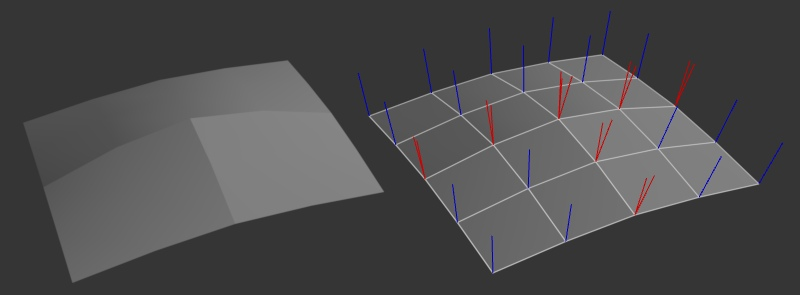
We can average multiple normals together to make the surface look smooth and much more detailed than it actually is. Now, several faces share the same vertex normal which is used to calculate the shading on the surface.



vertex normals in a smoothed mesh

Or another way, we define 3 smooth areas with hard edges between them. You can see on the right that the vertices on those edges still have some of their vertex normals diverging from another.

We wouldn't want to rotate and average dozens or hundreds of normals ourselves so we usually use smoothing groups or define hard edges to control the smoothness of our surfaces. Our 3d application will then average normals based on those rules. There are, however, cases where we might want to define our vertex normals explicitly.

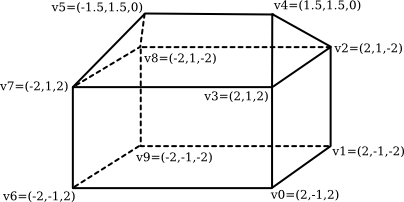


smoothing groups

# Indexed Face Sets[[10]](#footnote-9)

The polygons in a polygonal mesh are also referred to as "faces" (as in the faces of a polyhedron), and one of the primary means for representing a polygonal mesh is as an indexed face set, or IFS.

The data for an IFS includes a list of all the vertices that appear in the mesh, giving the coordinates of each vertex. A vertex can then be identified by an integer that specifies its index, or position, in the list. As an example, consider this "house," a polyhedron with 10 vertices and 9 faces:



The vertex list for this polyhedron has the form

Vertex #0. (2, -1, 2)

Vertex #1. (2, -1, -2)

Vertex #2. (2, 1, -2)

Vertex #3. (2, 1, 2)

Vertex #4. (1.5, 1.5, 0)

Vertex #5. (-1.5, 1.5, 0)

Vertex #6. (-2, -1, 2)

Vertex #7. (-2, 1, 2)

Vertex #8. (-2, 1, -2)

Vertex #9. (-2, -1, -2)

The order of the vertices is completely arbitrary (can be changed if needed). The purpose is simply to allow each vertex to be identified by an integer.

To describe one of the polygonal faces of a mesh, we just have to list its vertices, in order going around the polygon. For an IFS, we can specify a vertex by giving its index in the list. For example, we can say that one of the triangular faces of the pyramid is the polygon formed by vertex #3, vertex #2, and vertex #4. So, we can complete our data for the mesh by giving a list of vertex indices for each face. Here is the face data for the house. Remember that the numbers in parenthese are indices into the vertex list:

Face #0: (0, 1, 2, 3)

Face #1: (3, 2, 4)

Face #2: (7, 3, 4, 5)

Face #3: (2, 8, 5, 4)

Face #4: (5, 8, 7)

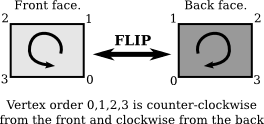
Face #5: (0, 3, 7, 6)

Face #6: (0, 6, 9, 1)

Face #7: (2, 1, 9, 8)

Face #8: (6, 7, 8, 9)

Again, the order in which the faces are listed in arbitrary. There is also some freedom in how the vertices for a face are listed. You can start with any vertex. Once you've picked a starting vertex, there are two possible orderings, corresponding to the two possible directions in which you can go around the circumference of the polygon. For example, starting with vertex 0, the first face in the list could be specified either as (0,1,2,3) or as (0,3,2,1). However, the first possibility is the right one in this case, for the following reason. A polygon in 3D can be viewed from either side; we can think of it as having two faces, facing in opposite directions. It turns out that it is often convenient to consider one of those faces to be the "front face" of the polygon and one to be the "back face." For a polyhedron like the house, the front face is the one that faces the outside of the polyhedron. The usual rule is that the vertices of a polygon should be listed in counter-clockwise order when looking at the front face of the polygon. When looking at the back face, the vertices will be listed in clockwise order. This is the default rule used by OpenGL.

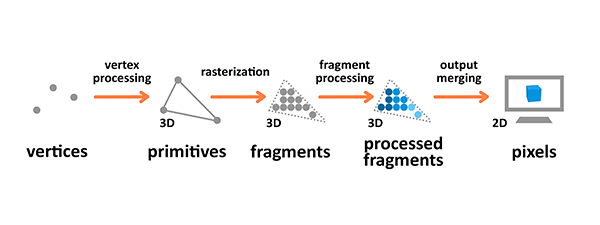


## 3D file formats[[11]](#footnote-10)

The basic purpose of a 3D file format is to store information about 3D models as plain text or binary data. In particular, they encode the 3D model’s *geometry*, *appearance*, *scene*, and *animations*. There are a lot of 3d formats like (fbx, 3ds, blend, obj, …etc). FBX is one of the most used 3d formats.

# Rendering pipeline

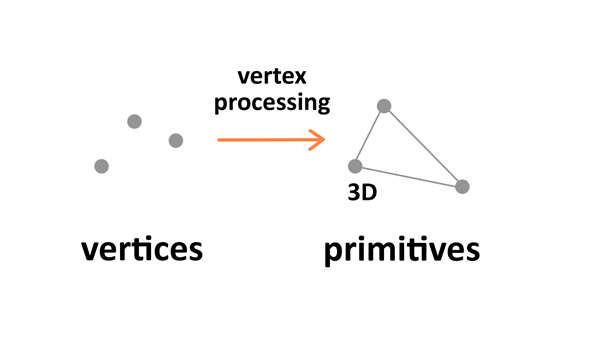
The rendering pipeline is the process by which images are prepared and output onto the screen. The graphics rendering pipeline takes the 3D objects built from **primitives** described using **vertices**, applies processing, calculates the **fragments** and renders them on the 2D screen as **pixels**.



Terminology used in the diagram above is as follows:

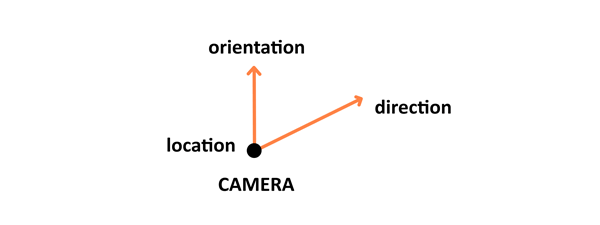
* A **Primitive**: An input to the pipeline — it's built from vertices and can be a triangle, point or line.
* A **Fragment**: A 3D projection of a pixel, which has all the same attributes as a pixel.
* A **Pixel**: A point on the screen arranged in the 2D grid, which holds an RGBA (Red-Green-Blue- Alpha (Transparency)) color.

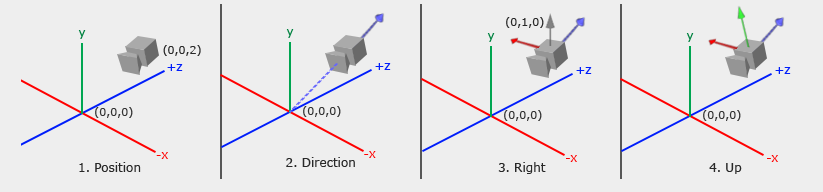
Vertex processing is about combining the information about individual vertices into primitives and setting their coordinates in the 3D space for the viewer to see. It's like taking a photo of the given scenery you have prepared — you have to place the objects first, configure the camera, and then take the shot.



There are four stages to this processing:

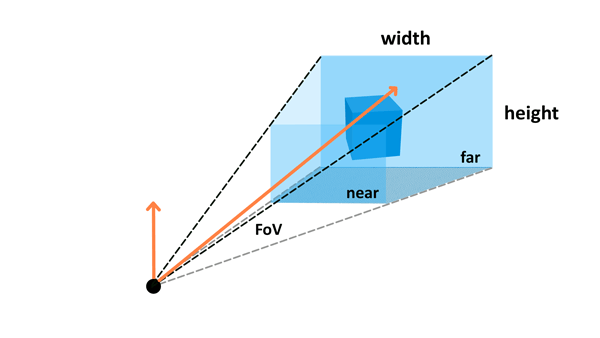
1. **Model transformation**:the first one involves arranging the objects in the world.
2. **view transformation** which takes care of positioning and setting the orientation of the **camera** in the 3D space. The camera has three parameters — **location, direction, and orientation** — which have to be defined for the newly created scene.

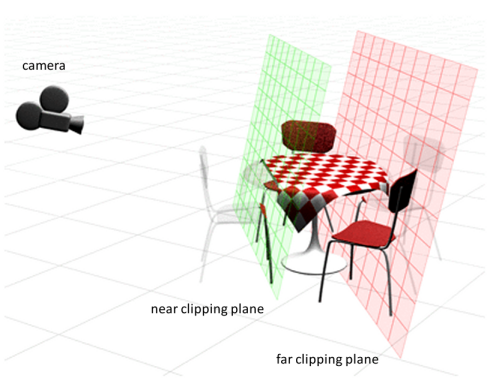




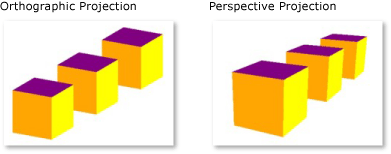
1. **Projection transformation**: then defines the camera settings. It sets up what can be seen by the camera. It contains 2 types: perspective and orthographic.
   1. **Perspective View**: the configuration includes *field of view*, *aspect ratio* and optional *near* and *far planes*.

A perspective camera is how we see the real world. If we take a look at the things around us, they have depth and we can judge their distance. Imagine looking at a very long road. It will appear to get narrower as it goes further into the distance. This is due to perspective.





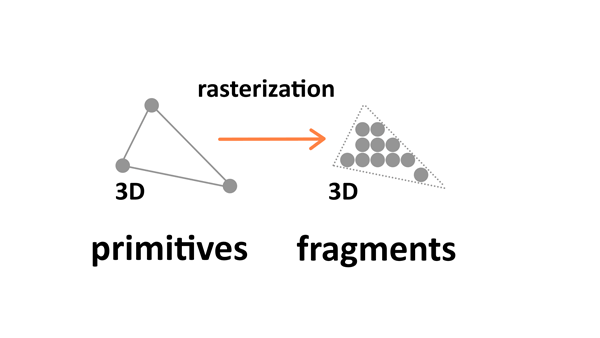
* 1. An **orthographic camera** however removes this sense of perspective. Objects are drawn without perspective distortion. See the image below:



1. The last step is **viewport transformation**, which involves outputting everything for the next step in the rendering pipeline.

## Rasterization

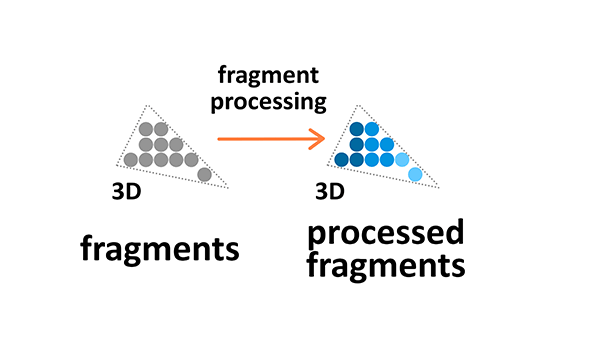
Rasterization converts primitives (which are connected vertices) to a set of fragments.



Those fragments — which are 3D projections of the 2D pixels — are aligned to the pixel grid, so eventually they can be printed out as pixels on a 2D screen display during the output merging stage.

## Fragment processing

Fragment processing focuses on textures and lighting — it calculates final colors based on the given parameters.



# Textures

Textures are 2D images used in the 3D space to make the objects look better and more realistic. Textures are combined from single texture elements called texels the same way picture elements are combined from pixels. Applying textures onto objects during the fragment processing stage of the rendering pipeline allows us to adjust it by wrapping and filtering it if necessary.

Texture wrapping allows us to repeat the 2D image around the 3D object. Texture filtering is applied when the original resolution or the texture image is different from the displayed fragment — it will be minified or magnified accordingly.



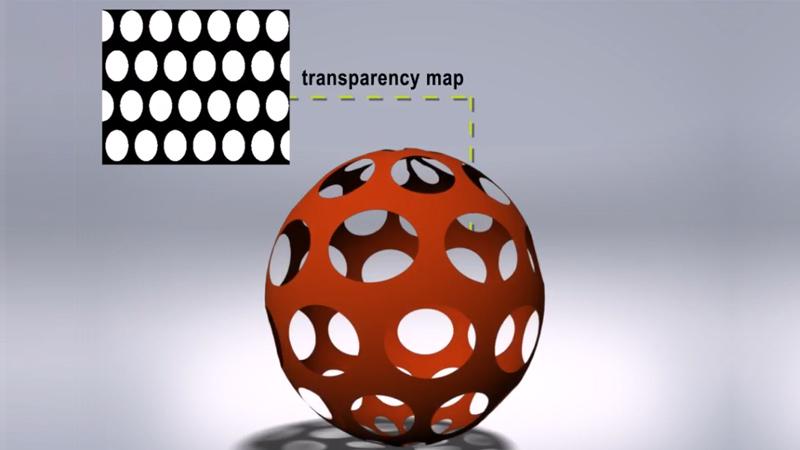
## Types of texture Maps

When texture mapping first came about in the 1970s, the original texture maps was created mainly to wrap a bitmap image around a 3D object, helping to save time when designing 3D models.

Bump mapping was subsequently invented to create the illusion of bumpiness and depth in objects to increase the realism in them. Over the years, computer scientists came up with more and more types of texture maps to enhance the realism of 3D objects.

## This is a list of texture maps with brief descriptions[[12]](#footnote-11):

* **Color map**, **diffuse** map, or **Albedo** map: Gives the object colours, or an appearance that you are aiming for.
* **Metalness** map: Tells you the metallic properties of the material. Materials are either dielectric or conductive (non-metals and metals).
* **Roughness** map or gloss map: Tells you the glossiness of the surface, determines how reflective the material is.
* **Transparency Map:** grey scale textures that use black and white values to signify areas of transparency or opacity on an object's material



* **Bump** map: Creates the illusion of depth by faking اصطناع بدون تغيير فعلي details on the surface using height information to tell when a point should be up or down. The surface however is still flat and not actually warped.



* **Normal** map: A type of bump map that fakes the details by using angle information to tell which direction each point of the surface should be oriented towards. This gives better effect with lighting and shadows.



* **Displacement** map: Similar to a height map but actually changes the geometry of the object by moving vertices (points) on the mesh (surface). This physically alters the form of the object.



**Normal Maps VS. Bump Maps**

both affect the normals of your geometry and create the illusion of detail without having to rely on extra geometry

**bump maps** just encode height information using black and white values

**Black**: minimum height delta

**White**: maximum height delta

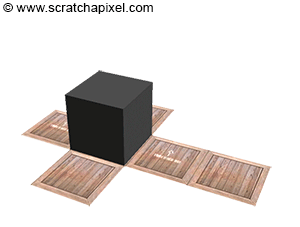
**normal maps** use RGB values to signify the orientation of the surface normals:

 red, green and blue channels = X, Y and Z orientation of the surface

Normal maps can typically get **more detailed** information onto the surface

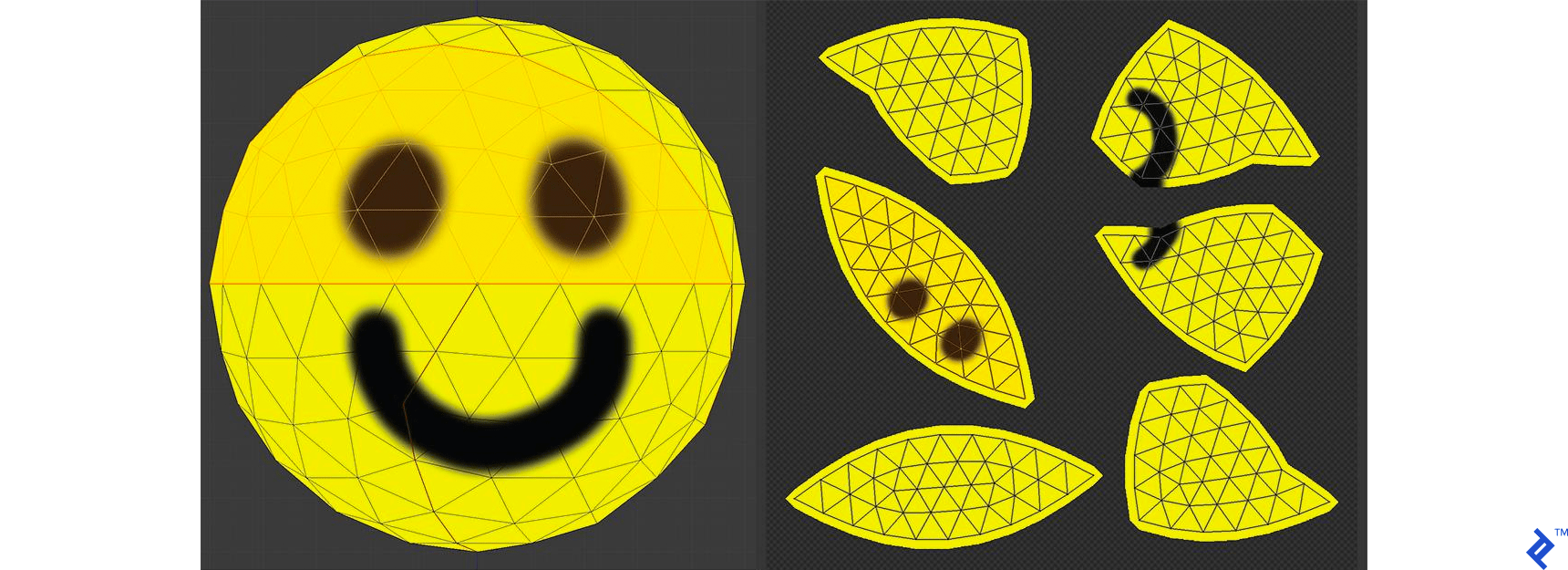
### Texture Coordinates

The last significant property are texture coordinates, commonly referred to as UV mapping. You have a model, and a texture that you want to apply to it. The texture has various areas on it, representing images that we want to apply to different parts of the model. There has to be a way to mark which triangle should be represented with which part of the texture. That’s where texture mapping comes in.



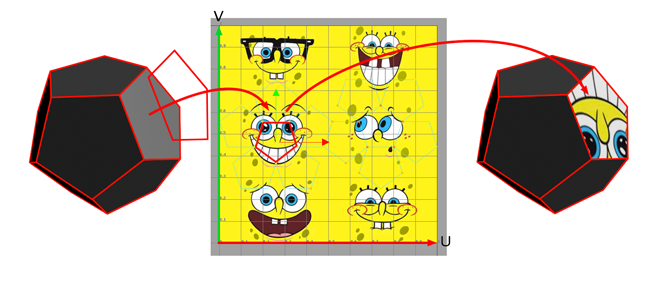
For each vertex, we mark two coordinates, U and V. These coordinates represent a position on the texture, with U representing the horizontal axis, and V the vertical axis. The values aren’t in pixels, but a percentage position within the image. The bottom-left corner of the image is represented with two zeros, while the top-right is represented with two ones.

A triangle is just painted by taking the UV coordinates of each vertex in the triangle, and applying the image that is captured between those coordinates on the texture.

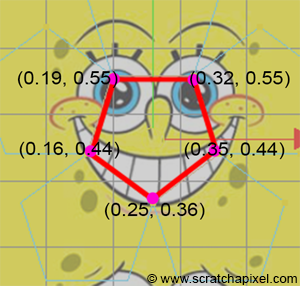


You can see a demonstration of UV mapping on the image above. The spherical model was taken, and cut into parts that are small enough to be flattened onto a 2D surface. The seams where the cuts were made are marked with thicker lines. One of the patches has been highlighted, so you can nicely see how things match. You can also see how a seam through the middle of the smile places parts of the mouth into two different patches.

The wireframes aren’t part of the texture, but just overlayed over the image so you can see how things map together.



A face of the mesh is applied on the surface of a square image. We apply the face on the surface of our digital wallpaper, use its shape to cut out a piece of the wallpaper which we then apply to the surface of the face on the 3D model. This is the principle of texture mapping. The uv-coordinates of the face, are 2D coordinates which define the position of the face on the surface of the image with respect to the uv-space coordinate system whose origin is in the bottom-left coordinate of the image.



## 2D texture coordinates defined with respect to a Cartesian coordinate system whose origin is located in the lower-left corner of the square image

## Lighting

The colors we see on the screen is a result of the light source interacting with the surface colors of the object's material. Light might be absorbed or reflected. There are four common basic types of lighting:

* **Diffuse**: A distant directional light, like the sun.
* **Specular**: A point of light, just like a light bulb in a room or a flash light.
* **Ambient**: The constant light applied to everything on the scene.
* **Emissive**: The light emitted directly by the object.

## Types of 3d light sources[[13]](#footnote-12)

### Directional Light

Directional lights are most often used to simulate **sunlight** and **moonlight**. Typically functioning as a key light in a scene, they provide controllable and predictable illumination. Due to the fact that the sun is so far away from the earth, by the time the light reaches us, the rays are essentially parallel to one another . That is why you will also hear the term *parallel light* when referring to directional lights.

**A typical representation of a 3D directional light.**

### Ambient Light

Ambient light is light that is spread everywhere equally in **all directions** without dissipating with distance. Although no true ambient light exists in the real world, its general purpose in 3D is to simulate the bounced light that occurs all around us. Essentially

**An ambient light.**

### Spotlights

One of the most commonly used lights in 3D is a spotlight because of the amount of control the artist has over its parameters and variety of effects. Spotlights are often the key light in a scene as well.

**A spotlight.**

### Point or Omni Lights

Just like it sounds, a point light emits from a single point in all directions. Also referred to as a uniform light, it is ideal for lightbulbs, lamps, candles, and the like. Point lights often are used for small areas like rooms.

**A point light.**

You will find many additional types of lights in various 3D packages. Each is a variation of the core types discussed here. Each is designed to serve specific purposes. You will encounter lights such as area lights, volume lights, skylights, and sunlight systems. Play around with each of them and you'll quickly discover their individual purposes even though their functionality doesn't stray much from the light types discussed here

## Output merging

During the output manipulation stage all the fragments of the primitives from the 3D space are transformed into a 2D grid of pixels that are then printed out on the screen display.



During output merging some processing is also applied to ignore information that is not needed — for example the parameters of objects that are outside of the screen or behind other objects, and thus not visible, are not calculated.

### Framebuffer[[14]](#footnote-13)

The destination of a rendering pipeline is a Framebuffer. A framebuffer contains several attachments such as Color and Depth . However, a framebuffer can display the rendering content to a screen ONLY if a 2D array memory has been allocated and attached to a framebuffer attachment. A 2D array memory is known as a Texture image.

### Light and texture baking[[15]](#footnote-14)

Baking is a term that is used widely in the 3D community. It is a term that can be applied to many different processes. What it generally means is, freezing and recording the result of a computer process. It is used in everything from animations, to simulations, to texturing 3d models and much more.

There are 2 kinds of lights that can be used for games: Static and Dynamic[[16]](#footnote-15). Dynamic lights interactively respond to the 3D scene, like shifting shadows and informing the materials they touch. Static lights are stationary and can be excluded from dynamic calculation to save game resources

# Three Dimensional Transformations

The geometric transformations play a vital role in generating images of three Dimensional objects with the help of these transformations. The location of objects relative to others can be easily expressed. Sometimes viewpoint changes rapidly, or sometimes objects move in relation to each other. For this number of transformation can be carried out repeatedly.

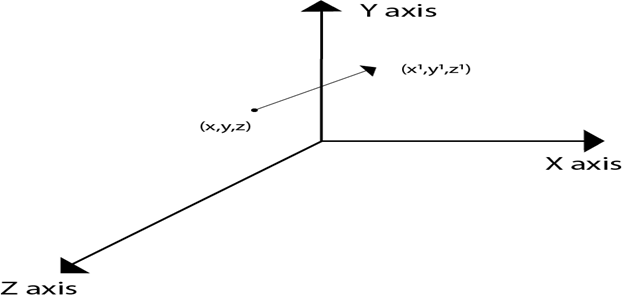
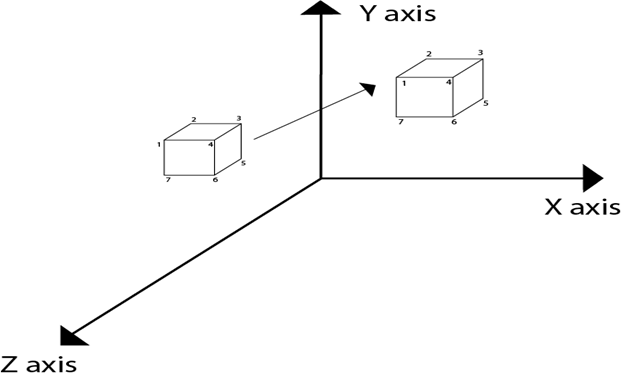
# Translation

It is the movement of an object from one position to another position. Translation is done using translation vectors. There are three vectors in 3D instead of two. These vectors are in x, y, and z directions. Translation in the x-direction is represented using Tx. The translation is y-direction is represented using Ty. The translation in the z- direction is represented using Tz.

If P is a point having co-ordinates in three directions (x, y, z) is translated, then after translation its coordinates will be (x1 y1 z1) after translation. Tx Ty Tz are translation vectors in x, y, and z directions respectively.

        x1=x+ Tx  
          y1=y+Ty  
          z1=z+ Tz

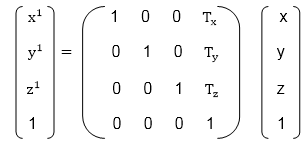
Three-dimensional transformations are performed by transforming each vertex of the object. If an object has five corners, then the translation will be accomplished by translating all five points to new locations. Following figure 1 shows the translation of point figure 2 shows the translation of the cube.

## Matrix for translation

## Matrix representation of point translation

Point shown in fig is (x, y, z). It become (x1,y1,z1) after translation. Tx Ty Tz are translation vector.

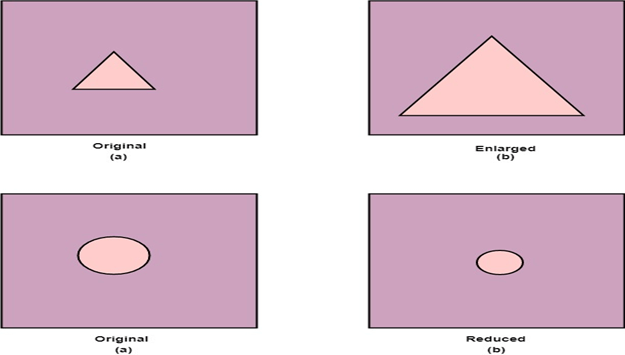


****

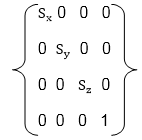
# Scaling

Scaling is used to change the size of an object. The size can be increased or decreased. The scaling three factors are required Sx Sy and Sz.

Sx=Scaling factor in x- direction  
Sy=Scaling factor in y-direction  
Sz=Scaling factor in z-direction



## Matrix for Scaling



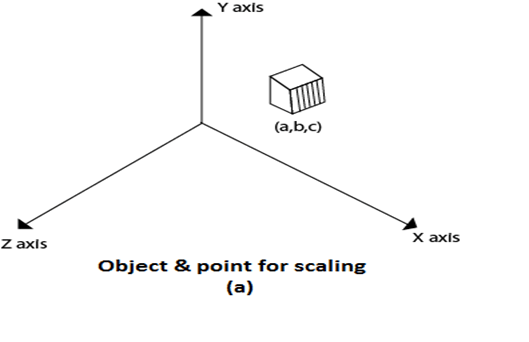
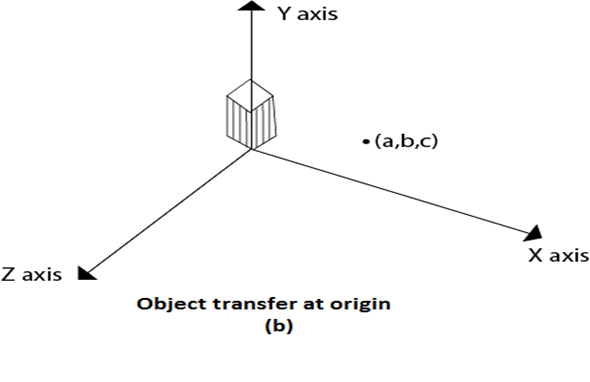
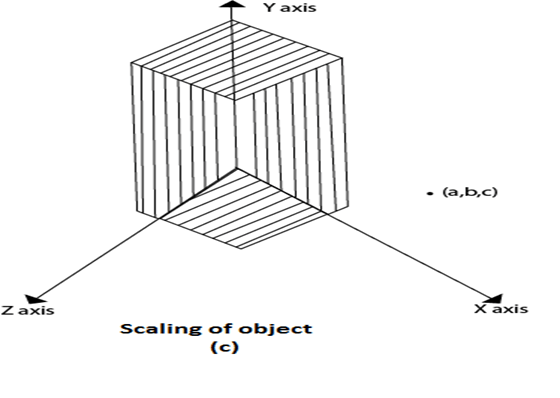
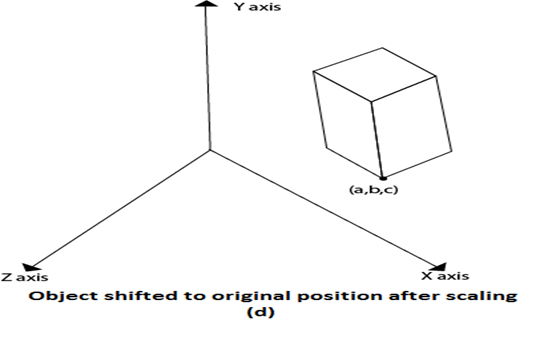
## Scaling of the object relative to a fixed point

Following are steps performed when scaling of objects with fixed point (a, b, c). It can be represented as below:

1. Translate fixed point to the origin
2. Scale the object relative to the origin
3. Translate object back to its original position.

#### Note: If all scaling factors Sx=Sy=Sz.Then scaling is called as uniform. If scaling is done with different scaling vectors, it is called a differential scaling.

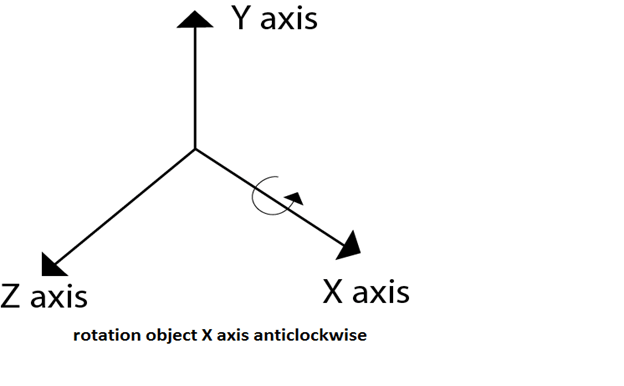
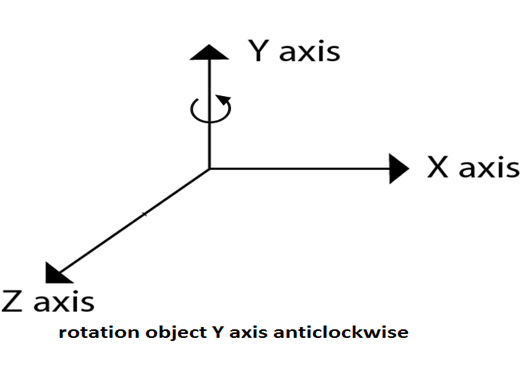
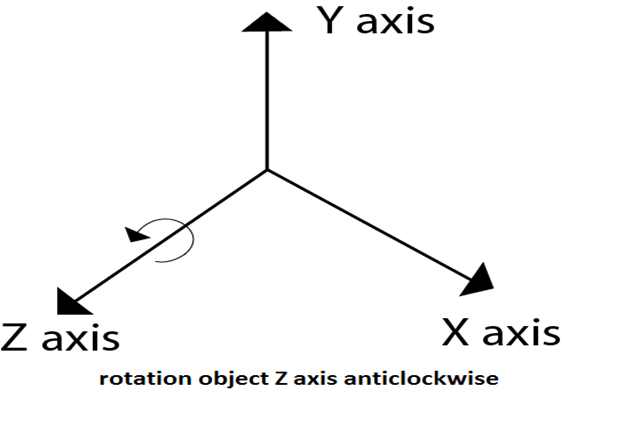
In figure (a) point (a, b, c) is shown, and object whose scaling is to done also shown in steps in fig (b), fig (c) and fig (d).

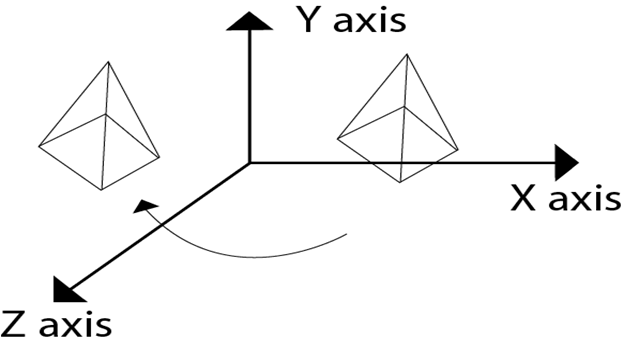
# Rotation

It is moving of an object about an angle. Movement can be anticlockwise or clockwise. 3D rotation is complex as compared to the 2D rotation. For 2D we describe the angle of rotation, but for a 3D angle of rotation and axis of rotation are required. The axis can be either x or y or z.

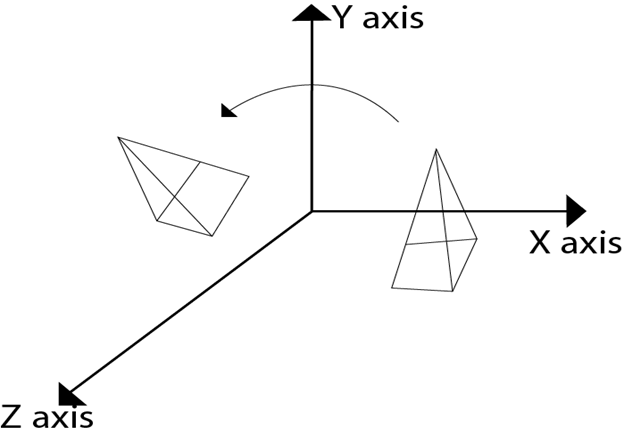
**Following figures shows rotation about x, y, z- axis**

Following figure show rotation of the object about the Y axis



Following figure show rotation of the object about the Z axis

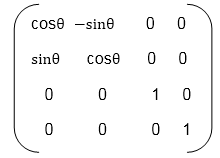


# Rotation about Arbitrary Axis

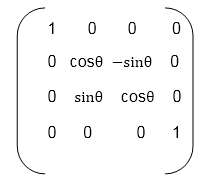
When the object is rotated about an axis that is not parallel to any one of co-ordinate axis, i.e., x, y, z. Then additional transformations are required. First of all, alignment is needed, and then the object is being back to the original position. Following steps are required

1. Translate the object to the origin
2. Rotate object so that axis of object coincide with any of coordinate axis.
3. Perform rotation about co-ordinate axis with whom coinciding is done.
4. Apply inverse rotation to bring rotation back to the original position.

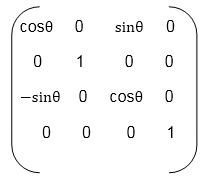
## Matrix for representing three-dimensional rotations about the Z axis



## Matrix for representing three-dimensional rotations about the X axis



## Matrix for representing three-dimensional rotations about the Y axis



# 3D Modelling Techniques [[17]](#footnote-16)

3D models can be created in many different ways. The choice of modelling technique depends on the requirements and the complexity of the object. The following list describes some of popular 3D modelling techniques:

## Standard Primitives and Modifiers:

|  |
| --- |
| 3d modelling Primitives and Modifiers |
| Primitives and Modifiers |

Many 3D software packages include tools for creating standard objects such as boxes and spheres easily. One of the simplest 3D modelling techniques is to combine these standard objects to create complex 3D models. 3D Studio MAX includes standard objects such as sphere, cube and tube. These standard objects can be modified through their parameters (radius, height etc.) or through special modifiers (stretch, bend etc.). By combining several different standard objects and by modifying them, one can create complex 3D models.

## Polygon Modelling:

|  |
| --- |
| Stages of Polygon Modelling |
| Stages of Polygon Modelling |

Complex objects are often modeled polygon by polygon. 3D software packages include many efficient tools for creating and manipulating polygons.

Subdivision surface means a surface which is created by dividing the original 3D model into smaller polygons. At the same time 3D model's corners become rounder and the surface becomes smoother. Subdivision surfaces is a very popular modelling technique. The advantage of a subdivision surface is the fact that one can create a coarse 3D model which is then automatically subdivided into a smoother surface.

|  |
| --- |
| 3d modelling |
| A Polygon Model Before and After Subdivision |

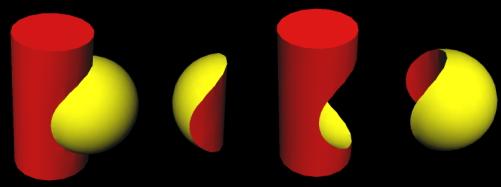
## Boolean Operations:

The starting point of Boolean operation is two overlapping 3D objects. Boolean operations are prone to error and the resulting geometry might have underlying problems.

|  |
| --- |
| 3d boolean modelling |
| Boolean Modelling |

#### Boolean operation has four possible results:

|  |
| --- |
| The 4 Possible Outcomes of a Boolean Operation |
| The 4 Possible Outcomes of a Boolean Operation |

* **Union:** Two 3D models are combined and the unnecessary geometry inside of the Models is removed.
* **Intersection:** Overlapping a part of the two 3D objects.
* **Subtraction (A-B):** Object A is subtracted from object B.
* **Subtraction (B-A):** Object B is subtracted from object A.

#### https://upload.wikimedia.org/wikipedia/commons/8/8b/Csg_tree.pnghttp://meshmixer.com/forum/index.php?action=dlattach;topic=468.0;attach=201

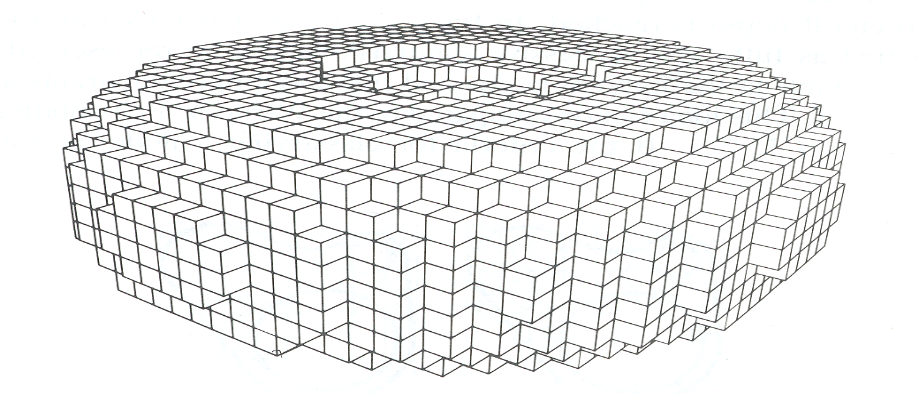
## NURBS:

### NURBS stands for **Non-Uniform Rational B-spline**. In NURBS modelling, lines and surfaces are not manipulated by moving vertices, edges, faces or polygons. Instead NURBS surfaces and lines are manipulated by special control points.

|  |
| --- |
| A cross section and the object created by a solid of revolution |
| A cross section and the object created by a solid of revolution |

## Voxel



This technique is to Partition space into uniform grid Grid cells are called voxels(like pixels). 

This is a simple technique however it has a problem of storage for voxels of side length of *n* . Which can be O(*n*3) storage for NxNxN grid, (for example when n=1000, 1 billion voxels for 1000x1000x1000 )



1. <https://www.javatpoint.com/computer-graphics-3d-graphics> [↑](#footnote-ref-0)
2. <https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_on_the_web/Basic_theory> [↑](#footnote-ref-1)
3. <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/bb324490(v=vs.85)> [↑](#footnote-ref-2)
4. <https://www.haroldserrano.com/blog/before-using-metal-computer-graphics-basics> [↑](#footnote-ref-3)
5. <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh> [↑](#footnote-ref-4)
6. <https://mastersketchup.com/introduction-to-quad-modeling/> [↑](#footnote-ref-5)
7. <https://www.racoon-artworks.de/cgbasics/normals.php> [↑](#footnote-ref-6)
8. <https://www.toptal.com/javascript/3d-graphics-a-webgl-tutorial> [↑](#footnote-ref-7)
9. <https://mastersketchup.com/introduction-to-quad-modeling/> [↑](#footnote-ref-8)
10. <http://math.hws.edu/graphicsbook/c3/s4.html> [↑](#footnote-ref-9)
11. <https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/> [↑](#footnote-ref-10)
12. <https://chunaik.medium.com/texture-mapping-in-3d-6dffd54d3a54> [↑](#footnote-ref-11)
13. <https://www.peachpit.com/articles/article.aspx?p=174370&seqNum=4> [↑](#footnote-ref-12)
14. <https://www.haroldserrano.com/blog/before-using-metal-computer-graphics-basics> [↑](#footnote-ref-13)
15. <http://handlebar3d.com/what-is-baking> [↑](#footnote-ref-14)
16. <https://cgcookie.com/articles/big-idea-baking> [↑](#footnote-ref-15)
17. <https://www.onlinedesignteacher.com/2014/07/3d-modelling-basics_33.html> [↑](#footnote-ref-16)