1. **Scalars, Points, and Vectors**

- **Scalars** are real numbers used to represent quantities like distance or angle. They follow arithmetic rules (addition, multiplication, etc.).
- **Points** represent locations in 3D space, with no size or shape. They're defined only by their position.
- **Vectors** have both magnitude and direction (like velocity). They’re often represented as directed line segments between points.

Example: In a 3D graphics program, `P - Q` represents a vector pointing from point Q to point P.
Possible Quiz Answer: "A vector is different from a point because it has direction and magnitude, not just position."

2. **Geometric Objects and Operations**

- You can **add vectors** using the head-to-tail rule.
- **Scalar multiplication** with a vector changes its length but not its direction.
- **Zero vector** has no length or defined direction (e.g., sum of two opposite vectors).

Example: If `A` is a vector with length 3 in a certain direction, `2A` will have length 6 in the same direction.
Possible Quiz Answer: "Vector addition results in a new vector that follows the head-to-tail rule for combining magnitudes and directions."

3. **Coordinate-Free Geometry**

- Geometric objects can be defined without a coordinate system; relative positions and shapes remain the same even if you remove the axes.

Example: A square remains a square regardless of the coordinate system; it’s identified by its shape, not by coordinates.
Possible Quiz Answer: "Coordinate-free geometry means geometric relationships are independent of the specific coordinate values."

4. **Mathematical View: Vector, Affine, and Euclidean Spaces**

- **Vector space** includes vectors and scalars.
- **Affine space** includes points in addition to vectors, allowing operations like point-vector addition.
- **Euclidean space** adds measurements (distances) to vectors.

Example: In an affine space, the point `P` from point `Q` with displacement `v` can be written as `P = Q + v`.
Possible Quiz Answer: "Affine space allows operations like adding a vector to a point to yield a new point."

5. **Computer Science View**

- Scalars, points, and vectors can be treated as abstract data types (ADTs) in programming, allowing operations like addition to be implemented without worrying about internal data representation.

Example: In C++, defining `point` and `vector` types lets you write code like `q = p + a * v`.
Possible Quiz Answer: "In computer graphics, abstract data types allow operations on geometric types independent of how they're implemented."

6. **Lines**

- **Parametric form** of a line: For a point (P) on a line, we use `P(α) = P0 + αd`, where `P0` is an initial point, `d` is a direction vector, and `α` determines the position along the line.

Example: The line passing through point (1,2,3) in direction (4,5,6) is given by `P(α) = (1,2,3) + α(4,5,6)`.
Possible Quiz Answer: "The parametric form of a line in 3D graphics allows us to define positions along the line by varying the scalar α."

7. **Affine Sums**

- **Affine sum**: Combines points with scalars to get a new point along the line defined by two points. This leads to expressions like `P = α1R + α2Q`, where `α1 + α2 = 1`.

Example: For points `Q` and `R`, the midpoint is `P = 0.5 * Q + 0.5 * R`.
Possible Quiz Answer: "Affine sums allow for interpolating between points, as in finding the midpoint of two points."

8. **Convexity**

- A **convex object** includes any point on the line segment between two points within it.
- The **convex hull** is the smallest convex object containing a set of points, like stretching a rubber band around them.

Example: For points arranged in a square, the convex hull is the square itself.
Possible Quiz Answer: "The convex hull of a shape is the smallest convex shape that can contain all points in the set."

9. **Dot and Cross Products**

- **Dot product**: Used to measure angles between vectors. If `u · v = 0`, the vectors are orthogonal.
- **Cross product**: Yields a vector orthogonal to both input vectors, useful for determining the normal of a surface.

Example: If vectors `u` and `v` represent two edges of a triangle, the cross product `u × v` is perpendicular to the triangle's surface.
Possible Quiz Answer: "The dot product of two vectors indicates if they are perpendicular; a result of zero means they are orthogonal."

4.2 Three-Dimensional Primitives

Understanding 3D graphics requires examining different types of shapes that extend beyond simple points, lines, and triangles. While these basic shapes work in both two and three dimensions, more complex shapes such as curves, surfaces, and volumes are also fundamental in 3D spaces. Here, the challenge lies in defining these shapes within a graphics system that can handle them effectively.

Dimensions in Space vs. Dimensions of Objects

When we talk about dimensions, we must consider two aspects:

1. **The dimension of the space** (e.g., 2D, 3D).

2. **The dimension of the object** within that space.

- **Points** are 0-dimensional because they represent a location without size.
- **Lines (or curves)** are 1-dimensional because they have length but no width.
- **Triangles or planes** are 2-dimensional because they have both length and width.

In 3D, objects can have up to three dimensions, encompassing points, lines, surfaces, and volumes.

Curves and Areas in 3D

Curves and areas add complexity:

- **Curves** can be approximated by connecting line segments, making them manageable in graphics systems.
- **Areas** (e.g., polygons) can be filled with colors, textures, or patterns. Complex polygons are typically broken down into triangles, which are simpler to render.

Moving from 2D to 3D

In 3D, we encounter additional object types, such as **curves in space**, **surfaces in space**, and **volumetric objects** like cubes or spheres. The key challenge in 3D graphics is ensuring objects are manageable for rendering while adhering to performance constraints.

Characteristics of Efficient 3D Graphics Objects

Most graphics systems support objects that follow these rules:

1. **Surface-Based**: 3D objects are usually described by their surfaces (think of a hollow shell).
2. **Vertex-Defined**: Objects are defined by sets of vertices in 3D space.
3. **Triangle-Compatible**: Objects should be composed of or approximated by triangles.

These rules optimize rendering efficiency since most graphics hardware is built to handle triangles at high speeds. Modern graphics cards can render millions of triangles per second, allowing for smooth, detailed images.

The Importance of Triangles

Triangles are fundamental in 3D graphics due to their simplicity. Each triangle is defined by three vertices and can lie flat in space, avoiding complications that arise with non-planar shapes. For objects with more vertices, like polygons, it's common practice to break them down (tessellate) into triangles for rendering.

Curved objects (e.g., spheres) are also approximated by triangle meshes. This mesh allows for realistic shading, lighting, and texturing, even though the shape is technically an approximation.

Constructive Solid Geometry (CSG)

An alternative modeling technique, **Constructive Solid Geometry (CSG)**, builds complex objects from basic volumes (e.g., spheres, cylinders) using operations like union and intersection. While this method is excellent for creating detailed models, it is more computationally intensive than surface-based rendering.

In this section, the focus shifts from viewing vectors and points as abstract objects to practical representations in specific coordinate systems, particularly in 2D, 3D, and 4D spaces. This is essential in computer graphics, where transformations like rotation, translation, and scaling of objects require a clear coordinate system to map out an object's location and orientation accurately.

4.3.1 Coordinate Systems and Frames

A coordinate system is defined by a basis, which is a set of linearly independent vectors in a given space. In a 3D vector space, any vector can be represented as a unique combination of three basis vectors. These vectors are the building blocks for describing any vector in the space. For example, any 3D vector (v) can be written as: [v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3] where (\alpha_1), (\alpha_2), and (\alpha_3) are the components of (v) relative to the basis vectors (v_1), (v_2), and (v_3).

For practical applications, especially when working with points, we use a **frame**. A frame includes both an origin point and the basis vectors. With a frame, we can represent both points and vectors effectively:

- A vector's representation in a frame needs three components.
- A point's representation requires the origin's location, which is defined within the frame.

4.3.2 Change of Coordinate Systems

Often in graphics, we change coordinate systems to view objects from different perspectives, like the object frame, world frame, or camera frame. When transforming a vector from one coordinate system to another, we use a transformation matrix. For instance, given two bases, say ({ v_1, v_2, v_3 }) and ({ u_1, u_2, u_3 }), the transformation matrix (M) represents how each (u)-basis vector can be written in terms of the (v)-basis vectors.

4.3.4 Homogeneous Coordinates

To differentiate between points and vectors in 3D, we introduce **homogeneous coordinates**, which add a fourth component:

- For points, we set this component to 1.
- For vectors, we set it to 0.

Using homogeneous coordinates allows us to perform transformations (like translation) that wouldn't be possible in a 3D-only coordinate system. This 4D representation simplifies the matrix calculations needed for complex transformations and allows for seamless concatenation of multiple transformations in graphics applications.

4.3.5 Example: Change in Frames

An example of frame transformation illustrates how a vector represented in one basis is converted to another basis. Using the transformation matrix from the previous section, we can calculate the vector's new representation in the updated frame. This transformation enables graphics applications to shift smoothly between object, world, and camera coordinates.

In summary, understanding coordinate systems, frames, and homogeneous coordinates is fundamental to manipulating objects in computer graphics. They provide the structure for transforming objects, managing different viewing perspectives, and working in 3D space.

The fourth section of this chapter dives into **Frames in WebGL** and the transformation pipeline for handling vertex coordinates as they move from 3D models to 2D screens. This journey across various frames in WebGL involves applying multiple transformations to vertices, helping achieve the desired perspectives and projections on the display.

1. **Pipeline Frames and Coordinates**: WebGL uses a sequence of transformations to map vertices from their initial 3D positions in model coordinates to screen coordinates on a 2D display. Six frames are generally used:
    - **Model Coordinates**: Initial position of each object, defined by the application.
    - **Object (World) Coordinates**: Places all objects in a common world frame for consistent positioning.
    - **Eye (Camera) Coordinates**: Adjusts positions relative to the camera's view.
    - **Clip Coordinates**: Applies a projection to fit the view volume within a standardized space.
    - **Normalized Device Coordinates**: Normalizes coordinates for perspective division.
    - **Window (Screen) Coordinates**: Final 2D coordinates that match pixel positions on the screen.
2. **Model and Object Frames**: Objects are first placed in model coordinates, and then transformations are applied to position them in the world (object) frame. The world frame is where all objects are scaled, rotated, and placed together in relation to one another.
3. **Eye Coordinates**: The eye frame is defined by the camera, which has its origin at the camera's lens center, typically looking along the negative Z-axis. The transformation from model to world and then to eye coordinates is usually handled by the **model-view matrix**, a combination of transformations that simplifies positioning objects relative to the camera.
4. **Projection and Clip Coordinates**: After positioning in eye coordinates, a projection transformation (orthographic or perspective) is applied to map the 3D scene to a 2D screen. This produces clip coordinates, standardizing the viewing volume. Clip coordinates undergo a **perspective division** by the w-component to convert into normalized device coordinates.
5. **Final Screen Mapping**: Normalized device coordinates are then mapped to window (screen) coordinates using viewport settings, converting the 3D representation into 2D pixel values.

Practical Applications of Frame Transformations

Developers often utilize three main frames in WebGL: model, object (world), and eye frames. With the model-view matrix, developers can position objects with respect to the camera's view without manually altering individual vertex positions.

Example: Moving Objects in Front of the Camera

To position objects in front of the camera, one can apply a translation along the Z-axis in the model-view matrix rather than changing individual vertex coordinates. This is computationally efficient and maintains relative object orientations.

Example: Camera at an Angle

When positioning a camera at an angle, such as (1, 0, 1), pointing at the origin, transformations adjust the basis vectors for the new camera orientation. This requires an orthogonal coordinate system for the camera,

which can be created by calculating cross-products between camera orientation vectors.

Choosing Frame Transformation Approaches in WebGL

With WebGL's programmable shaders, developers have flexibility in where to apply transformations, either directly in the application or by offloading to the GPU via shader code. This choice impacts efficiency and performance, especially for complex scenes with numerous objects.

4.5 Matrix and Vector Types

In this section, we delve deeper into the types used in JavaScript for working with matrices and vectors in WebGL applications. Matrix types in the *MV.js* package include 3x3 and 4x4 matrices, while vector types can represent 2, 3, or 4 elements. These types are stored as one-dimensional arrays for efficient handling, but their structure is rarely needed directly, as a set of utility functions allows for manipulation.

For example, creating a 3-element vector can be done with:

```
var a = vec3();            // Creates a vec3 with all components set to 0
var b = vec3(1, 2, 3);     // Creates a vec3 with components 1, 2, and 3
var c = vec3(b);           // Copies the vec3 'b'
```

Similarly, 3x3 matrices are created with:

```
var d = mat3();            // Creates a 3x3 identity matrix
var e = mat3(0, 1, 2, 3, 4, 5, 6, 7, 8);  // Creates a mat3 with specified
elements
```

Since JavaScript lacks operator overloading (unlike GLSL or C++), operations such as vector addition and matrix multiplication are performed using dedicated functions:

```
a = add(b, c);       // Adds vectors 'b' and 'c'
d = mat4();          // Creates a 4x4 identity matrix
f = mult(e, d);      // Multiplies matrices 'e' and 'd'
```

These types are used similarly to GLSL types (e.g., `vec3` for 3D vectors), making it easier to transfer algorithms between application and shader code.

4.5.1 Row vs. Column Major Matrix Representation

Matrices in JavaScript and WebGL can be represented in either row-major or column-major order. Most JavaScript code, including the *MV.js* package, defaults to row-major order, but WebGL and OpenGL use column-major order. This discrepancy is managed by the `flatten` function, which converts matrices from row-major to column-major order for GPU processing.

4.6 Modeling a Colored Cube

With foundational knowledge of matrix and vector types, we can now create a 3D model—a rotating colored cube. Modeling this cube involves representing its vertices and faces using data structures compatible with WebGL.

4.6.1 Modeling the Faces

A cube can be represented by an array of vertices, with each face constructed from a sequence of these vertices:

```
var vertices = [
  vec3(-0.5, -0.5,  0.5), vec3(-0.5,  0.5,  0.5), vec3( 0.5,  0.5,  0.5), vec3(
0.5, -0.5,  0.5),
  vec3(-0.5, -0.5, -0.5), vec3(-0.5,  0.5, -0.5), vec3( 0.5,  0.5, -0.5), vec3(
0.5, -0.5, -0.5)
];
```

Each face is defined by a set of vertex indices, specifying the order of vertices for that face. The order is crucial for distinguishing front and back faces, with a counter-clockwise vertex order indicating an outward-facing side (according to the right-hand rule).

4.6.2 Inward and Outward Pointing Faces

Correct vertex ordering helps determine whether a face is "inward" or "outward" facing. This distinction allows us to cull back faces or apply different shading to front and back faces.

4.6.3 Data Structures for Object Representation

To represent the cube, we use a two-dimensional array structure for the vertices and faces. This separation of geometry (the vertices) from topology (the arrangement of faces) provides flexibility, allowing transformations to be applied more efficiently.

4.6.4 The Colored Cube

The `colorCube` function is used to assign colors to each face of the cube. Colors are chosen based on vertex indices, producing a vibrant representation of each face:

```
var colors = [
  [0.0, 0.0, 0.0, 1.0], [1.0, 0.0, 0.0, 1.0], [1.0, 1.0, 0.0, 1.0], [0.0, 1.0,
0.0, 1.0],
  [0.0, 0.0, 1.0, 1.0], [1.0, 0.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0], [0.0, 1.0,
1.0, 1.0]
];
```

The cube is rendered using two triangles per face, ensuring smooth coloring and accurate face depiction.

4.6.5 Color Interpolation

To fill in colors across each polygon face, we use color interpolation. The system interpolates between vertex colors based on barycentric coordinates, allowing for smooth transitions across surfaces.

4.6.6 Displaying the Cube

The cube display setup is similar to previous 3D examples, using shaders and orthographic projection.

These sections cover **Affine Transformations** (4.7) and fundamental transformations—**Translation, Rotation, and Scaling** (4.8)—essential concepts in computer graphics for manipulating geometric shapes.

4.7 Affine Transformations

Affine transformations are a type of transformation that map points or vectors from one location to another while preserving certain geometric properties. In computer graphics, these transformations allow us to change the shape, size, and position of objects without altering their structural relationships, which is essential for realistic rendering.

- **Transformations for Points and Vectors**: With affine transformations, both points and vectors can be represented as four-dimensional matrices in homogeneous coordinates. This allows us to apply transformations uniformly using matrix operations, simplifying computations.
- **Linearity**: Affine transformations are linear, meaning the transformed combination of points or vectors is equal to the combination of transformed points or vectors. This property is crucial because it means we only need to compute transformations for the endpoints of lines, and the intermediate points will follow, which is efficient for rendering.
- **Matrix Representation**: In homogeneous coordinates, affine transformations are represented by 4x4 matrices. The 12 elements in the matrix give the transformation 12 degrees of freedom, enabling complex manipulations like scaling, translation, and rotation.
- **Preservation of Lines**: Affine transformations maintain linearity, which ensures that lines remain straight. This feature is particularly useful for graphic systems, as only the endpoints of line segments need to undergo transformation, with the interior points generated during rasterization.

Affine transformations are widely used for creating realistic scenes by allowing transformations like translation, rotation, and scaling of objects while preserving their relative spatial relationships.

4.8 Translation, Rotation, and Scaling

These transformations—translation, rotation, and scaling—are the building blocks for manipulating objects in computer graphics. Each has unique properties that allow precise control over the position, orientation, and size of objects.

4.8.1 Translation

Translation moves every point of an object by a specific distance in a given direction. This transformation is defined by a displacement vector, which shifts the object without altering its shape or orientation. Translation has three degrees of freedom, corresponding to the three dimensions in space.

4.8.2 Rotation

Rotation turns an object around a specified axis. In 2D, rotation around the origin changes the angle of a point relative to the origin, while in 3D, we specify a rotation axis and an angle. The rotation matrix in 2D is given by:

[\begin{bmatrix} \cos \theta & -\sin \theta \ \sin \theta & \cos \theta \end{bmatrix}]

where (\theta) is the angle of rotation. In 3D, rotation is more complex and requires additional information to define the rotation axis.

Rotations have the following key characteristics:

1. **Fixed Point**: Rotations have a fixed point that remains unchanged, often the origin or the center of the object.
2. **3D Interpretation**: A 2D rotation can be visualized as a rotation in the (z=0) plane of 3D space, where each (z)-coordinate remains constant.
3. **Generalization**: In 3D, rotation requires defining an axis, angle, and a fixed point, with three degrees of freedom—two for the axis orientation and one for the rotation angle.

4.8.3 Scaling

Scaling changes the size of an object. **Uniform scaling** increases or decreases the size equally in all directions, while **non-uniform scaling** alters the object's size differently along each axis. This transformation is controlled by a scale factor, where:

- (\alpha > 1) enlarges the object,
- (0 < \alpha < 1) shrinks it, and
- (\alpha < 0) reflects the object around the origin.

Scaling has six degrees of freedom—three for the direction of scaling and three for specifying the fixed point, which remains unchanged.

Affine transformations in combination—translation, rotation, and scaling—provide a full set of tools for positioning, orienting, and sizing objects in a 3D space.

In Sections 4.9 and 4.10, we dive deep into **Transformations** within homogeneous coordinates and the powerful tool of **Concatenation of Transformations**. Here's a breakdown:

Section 4.9: Transformations in Homogeneous Coordinates

1. **Homogeneous Coordinates & Transformation Matrices**: In computer graphics, working with a 4x4 matrix representation for affine transformations (like translation, scaling, rotation, and shear) is standard. This is because adding a fourth homogeneous coordinate simplifies transformations and enables the use of matrix multiplication to combine transformations in a 3D space.

2. **Translation**: Moving a point from one position to another using a translation matrix that shifts the x, y, and z coordinates by a vector ((\alpha_x, \alpha_y, \alpha_z)).

3. **Scaling**: Adjusting the size of an object by scaling factors ((\beta_x, \beta_y, \beta_z)). The scaling matrix preserves the origin as a fixed point by default. If you want to scale around a different point, you need

to translate to the origin, scale, and then translate back.

4. **Rotation**: Involves rotation around the x, y, or z-axis by a specified angle. Matrix multiplications for rotation aren't commutative (order matters), so careful ordering of transformations is essential.

5. **Shear**: This transformation distorts the shape of an object. For example, an x-shear would shift the x-coordinates in relation to the y-coordinates while leaving y and z unchanged.

6. **Inverse Transformations**: Each transformation has an inverse (like translating by (-\alpha) to undo a previous (\alpha) translation) which is vital for undoing transformations or reversing sequences.

Section 4.10: Concatenation of Transformations

Concatenation allows multiple transformations (translation, rotation, scaling) to be combined in a single, efficient step. This strategy is essential in graphics systems where complex object transformations are needed.

1. **Applying Multiple Transformations**: Multiple transformations can be combined as a single matrix (M) which is applied to an object's points. For instance, instead of individually applying rotation, scaling, and translation, you can create one matrix by concatenating them, resulting in faster rendering, especially for large scenes.

2. **Rotation Around a Fixed Point**: Rotations are initially defined around the origin, but by translating an object to the origin, applying rotation, and translating back, you can rotate around any chosen point.

3. **Instance Transformations**: Used in rendering scenes with repeated objects, like trees in a forest. By defining a prototype, applying transformation matrices for different orientations, positions, and scales, graphics systems render the repeated objects efficiently with minimal data.

4. **Rotation About an Arbitrary Axis**: This advanced rotation technique allows for rotation around any axis and through any angle by aligning the axis with one of the primary axes, performing the rotation, and then re-aligning it back.

Practical Use in Graphics Pipelines

Graphics pipelines benefit significantly from concatenation because:

- **Efficiency**: By calculating and loading a single transformation matrix onto the GPU, thousands of points are transformed using just one matrix multiplication per point.
- **Reusability**: Transformations for standard rotations, scalings, or instance renderings can be reused, enhancing modularity and reducing complexity in rendering.