CS 3451: Computer Graphics Notes (Final Version)

17/10 Raytracing and Rendering

Why is raytracing slow?

- Nested for loops
    - For each pixel on screen
        - For each object
            - Does ray trace of object
- Ray tracing acceleration speeds this process up

Rendering a new primitive or shape requires several steps:

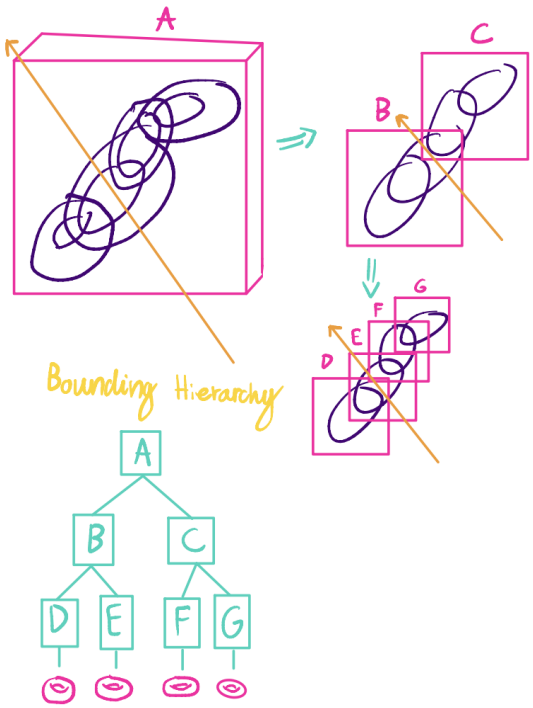
- First, the ray must intersect the object's surface.
- Following that, you need to determine the intersection point and the surface normal where the ray hits.
- Finally, you should determine the bounding box for the object.

Some objects are easier to raytrace, such as spheres, triangles, and cylinders, while others, like toruses, subdivided surfaces, cubic paths, fractals, "blobby spheres," and collections of polygons, are more challenging.

- Fast to ray trace
    - Sphere
    - Cylinder
    - Box
    - Triangle
    - Ellipsoid
- Slow to ray trace
    - Torus
    - cubic patches
    - fractals
    - Blobby spheres
    - Many triangles

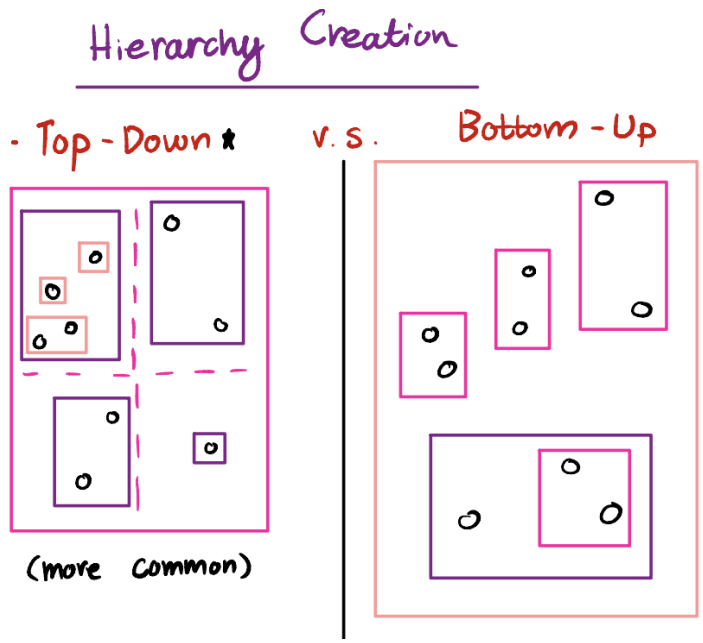
Bounding Volume

- To efficiently handle slow-to-raytrace objects, a bounding volume hierarchy is used.
- Bounding volumes (e.g., boxes, spheres, ellipsoids) are used to speed up raytracing by allowing for the elimination of objects that cannot be hit by the ray.
- Tight bounding volumes → fewer false positives
- Bounding hierarchies consist of nested bounding boxes, creating a tree-like structure.
    - Hierarchical traversal involves intersecting the ray with bounding volumes. If the bounding volume is hit, further checks are made with child nodes until the actual object is reached.



Constructing Bounding Hierarchy:

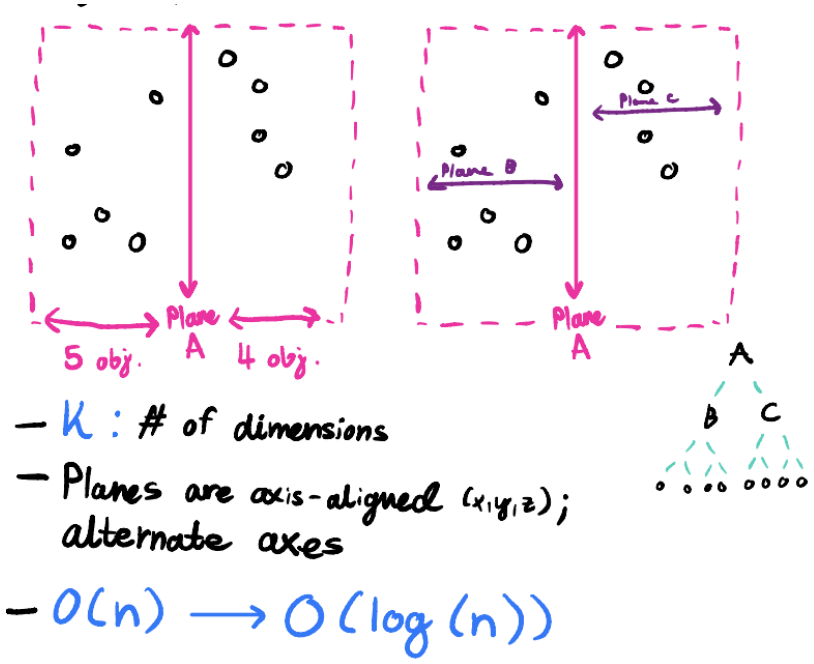
- Two common strategies for constructing bounding hierarchies are "bottom-up" and "top-down."
    - In the bottom-up strategy, pairs of objects are grouped into larger boxes iteratively, which ultimately leads to one giant box containing all objects.
    - In the top-down strategy, a single large bounding volume initially covers the entire scene. The goal is to iteratively divide this volume into smaller sub-volumes.
- The division process continues until each sub-volume contains a manageable number of objects, making the hierarchy tree more efficient for ray traversal.
- Achieving an optimal tree is difficult (NP-hard), but creating a sufficiently good tree is feasible.



Other Speed-Up Techniques:

- The GRIDS method involves dividing the scene into a 3D grid, with each cell containing a list of objects. During ray traversal, the algorithm checks each cell. If a cell is empty, it is skipped. If it contains objects, the algorithm checks for intersections with the objects.
- K-D trees are another technique for spatial partitioning, although the details are not provided in this text.

K-D Trees: subdivide space in half, with axis- aligned planes.



Raytracing in Industry:

- Raytracing is generally slower than rasterization but has become fast enough for various practical applications, particularly in the special effects industry.

Transition from Raytracing to Rasterization:

- The text mentions that they have completed their discussions on raytracing, and they will now shift their focus back to rasterization.

Creating Shadows without Raytracing:

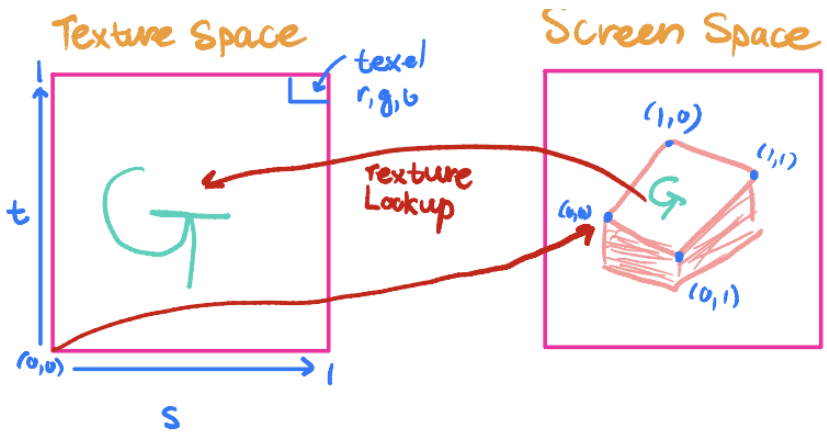
- Without raytracing, creating shadows in rasterization is a challenge. Two main techniques for raster-style shadows are "shadow volumes" and "shadow mapping," both of which are effective.
- Shadows are primarily about visibility; if light cannot "see" a surface, it's considered in shadow.
- Visibility checks are often handled using the Z-Buffer.
- Shadows can be "attached" (handled by shading equations) or "cast" (requiring visibility checks).

Shadow Mapping - Two-Pass Z-Buffer Method:

- The shadow mapping algorithm involves two passes: one from the perspective of the light source and one from the camera.
- The idea is to determine which pixels should be in shadow for the camera based on what was hidden from the light source.
- A pixel in the camera view is mapped from world space to both "light space" (with the mapping 'L') and "camera space" (with the mapping 'V').
- The algorithm compares the Z-position of the pixel in camera space to the Z-buffer value from the light's perspective. If the Z-position is less than the Z-buffer value, it is in shadow.
- Mapping details, especially how the mappings are derived and reversed, are not clearly explained in the text.
- For multiple light sources, each light source requires a separate render pass, potentially slowing down rendering.

Texture Mapping:

- The text introduces texture mapping as a technique to put textures on objects.
- Texture mapping involves mapping texture image coordinates onto a "texture space" defined by "texels" (texture pixels) and rendering the texture onto the screen.
- The process includes mapping and interpolating S,T (or UV) values across a polygon during rasterization, performing a "texture lookup" to find the color from the texture at that point, and coloring the pixel based on the texture color, in addition to lighting and shading.
- S,T coordinates typically range from 0 to 1 and are sometimes referred to as "UV" coordinates.



Interpolating Texture Coordinates:

- The text begins discussing the sub-stage of interpolating texture coordinates across a polygon during rasterization.
- It mentions a simple triangle polygon with texture coordinates (s1, t1), (s2, t2), (s3, t3).
- The goal is to interpolate these S-T (or UV) texture coordinates for a given point during rasterization, preparing for the next step.
- A hidden issue related to perspective projection is mentioned but deferred for discussion later.

Texture Mapping and Its Steps:

- Texture mapping involves three main steps: interpolating S/T values across a polygon, looking up the color at that point, and rendering the result.
- The text proceeds to delve into the details, beginning with interpolating S-T values.

Interpolating S-T Values:

- Texture interpolation, when dealing with perspective projection, is more complex than straightforward Gouraud interpolation.
- The following steps are involved:
    - Divide S/T by |Z| to obtain s' and t'.
    - Linearly interpolate s' and t' during rasterization, similar to Gouraud shading.
    - Divide by Z' on a per-pixel basis.
    - Perform a texture lookup to get the color of the textured surface.
- This process is necessary to avoid visual anomalies where textures appear to shift as the viewpoint changes.

Texture Lookup and Bilinear Interpolation:

- A direct texture lookup at S/T coordinates may result in a blocky, pixelated effect.
- To counter this, bilinear interpolation is used, where the color is calculated as a weighted average of the colors of the four nearest texels.
- This involves interpolating both horizontally and vertically between the relevant texels, effectively smoothing the transition.

Texture Magnification and Minification:

- Texture magnification is a situation where the blockiness of textures becomes noticeable, especially when close to an object.
- Texture minification, when the texture is far away and only a few pixels represent it, can result in problems such as sparkling or Moiré patterns.

Mip-Mapping and Image Pyramid:
- To address texture minification, multiple textures of different sizes are created, forming a "MIP-MAP" or image pyramid.
- Mip-mapping involves successively scaling down textures while averaging pixel colors.
- The image pyramid concept stacks different-sized images and uses these layers to calculate the texture color, leading to trilinear interpolation, which helps prevent the "sparkling effect."

Choosing Texture Size and Storage Efficiency:
- The choice of which size of texture to use depends on a function of the viewer's distance from the texture.
- To store multiple textures efficiently, an approach is described that involves dividing images into fourths and placing the original RGB values separately in boxes for each color channel.

Other Advanced Techniques:
- The text mentions that while mipmaps are widely supported and efficient, more elaborate methods exist for achieving higher-quality texture mapping.
- There are also various color interpolation techniques beyond bilinear and trilinear interpolation, aiming to improve performance and visual quality.

19/10

Bilinear Interpolation and Its Applications:
- Bilinear interpolation is commonly used in computer graphics not only for color interpolation but also for interpolating other attributes, such as heights.
- Suggests that bilinear interpolation of heights can create a smooth, curved surface that connects the height values.

Environment Mapping and Reflections:
- Environment mapping is introduced as a technique to simulate reflections in raster scenes, eliminating the need for ray tracing.
- Reflection in this context involves a ray "R" that hits a surface and bounces off.
    - Environment mapping simulates reflections by pasting photographs of the environment onto the inside of a giant sphere, making it look like the reflective surface.
- The technique involves taking a view vector from the eye to a point on the reflective surface, calculating the bounce direction using the normal vector, and then looking up the color from the environment map to obtain the reflected color.

    - While effective for simulating reflections, environment mapping is considered "fake" because it assumes all reflected objects are infinitely far away and flat, which may not be suitable for dynamic reflections or close-up views.
- Most implementations use a cube map instead of a sphere for environment mapping by rendering the scene from six different directions for the cube's faces.

Creating Bumpy Surfaces with Bump Maps:

- To give an object a bumpy appearance, one approach is to add many tiny polygons to the object, creating additional surface normals.
    - These extra polygons change how the object looks because they provide different surface normals.
- In 1978, Jim Blinn proposed an alternative method that approximates changes in surface normals using a technique called bump mapping.
- Bump mapping uses an image called a bump map that represents the "heights" of the object's surface.
    - By finding the slope/derivative of the surface defined by the bump map, a fake normal can be calculated and added to the object's real normal for a given pixel.
    - Bump mapping allows changes in shading using only an image.

24/10

Graphics Card Architecture and GLSL:

- Graphics cards process visual data and execute programs written in shading languages like GLSL and HLSL.
- In this class, the focus will be primarily on GLSL.
- Information flows through GLSL programs following a specific pipeline: input variables from OpenGL, vertex program, rasterizer, and fragment program.
- The vertex program takes input information for each vertex and outputs 2D vertex positions and colors for polygons. It usually passes texture coordinates unaltered.
- Bilinear color interpolation between front and back faces of a polygon is not commonly used.
- The rasterizer interpolates color and texture values across the polygon, creating fragments for each pixel.
- The fragment program operates on fragments, allowing manipulation of colors before rendering.

Introduction to GLSL:

- GLSL is a C-like programming language used to define graphic operations.
- It includes standard datatypes (int, float, bool) and graphics-specific datatypes (vec2, vec3, vec4, mat3, mat4, and uniform sampler2D for texture maps).
- The 'uniform' keyword signifies that the value is the same for all fragments, while 'varying' indicates values interpolated for each pixel.

- GLSL supports standard operators (+, -, *, %, =) and overloads some operators (e.g., * can multiply matrices, scale vectors, etc.).
- It offers various built-in functions like sin, cos, abs, floor, ceil, min, max, pow, sqrt, and vector operations (length, dot, cross, normalize).
- Unusual functions include inversesqrt and texture2D for specific purposes.

Example: Implementing a Diffuse Shader in GLSL:
- A diffuse shader is introduced as a teaching example.
- Vertex and fragment shaders are typically written together.
- The vertex program passes variables like normal and vertex_to_light, which will differ for each pixel.
- Predefined variables like gl_ModelViewProjectionMatrix and gl_NormalMatrix assist in transformations.
- The main function performs these transformations and calculates vectors.
- The fragment program is marked with #PROCESSING_COLOR_SHADER, indicating that it's a shader.
- Variables from the vertex shader are imported explicitly.
- It uses predefined variables like gl_FragColor for color output.
- The example implements a simple diffuse lighting model in GLSL.

Shaders and GLSL:
- Shaders play a significant role in modern graphics, and GLSL is a popular shading language used for this purpose.
- The graphics pipeline includes vertex shaders and fragment shaders, which work together to process and render polygons.
- Vertex shaders are used to modify 2D projections of 3D vertices and handle vertex manipulations on the GPU.
- Swizzling in GLSL allows for easy access to the components of a vector using dot notation (.xyzw, .rgba, .stpq).
- Example usage of swizzling is demonstrated, allowing the selection of vector components in groups or individually.

Vertex Shader Example: Twisting a Triangle:
- A vertex program is provided, which introduces a twist effect on a triangle.
- The program rotates the vertices counterclockwise based on their distance from the center of the screen, creating a shuriken-like, wavy shape.
- Vertex shaders are essential for handling vertex manipulations on the GPU, enabling interactive appearance changes for geometry.

Using Vertex and Fragment Shaders for Texture Mapping:
- A combination of vertex and fragment shaders is demonstrated for texture mapping.
- The vertex shader sets up texture coordinates that are interpolated by the rasterizer.
- The fragment shader uses the interpolated texture coordinates to sample a texture map and set the fragment color.

- This example illustrates the basic process of texturing objects.

Advanced Texture Processing: Combining Shifted Textures:
- An example of a more sophisticated texture processing program is provided.
- The vertex shader calculates two sets of texture coordinates, one shifted to the left and the other to the right.
- The fragment shader samples the texture map at both shifted coordinates and combines the colors to create a new texture.

Polygon Representation and Terminology:
- A brief discussion is provided on polygonal geometry and terminology.
- Polyhedrons are surfaces composed of polygons, where polygons are shapes made up of vertices and straight-line edges.
- Manifolds are portions of surfaces that look like (possibly bent) planes and can be smoothly traversed from any point to another.
- Manifold edges are edges entirely inside the manifold, while boundary edges are edges on the manifold. Non-manifold edges protrude from the plane of the manifold.
- Examples are given using the cube as a reference, illustrating different edge types.

26/10

Polygon Operations:
- Laplacian Smoothing is a technique used to smooth a polygonal surface by moving a vertex to the average position of its neighbors.
- Face Subdivision involves breaking a polygon into smaller faces, which can be used for smoothing rough surfaces.
- Triangulation is the process of converting a polygon into triangles, which are easier to work with.
- Storing connectivity information is essential for using these algorithms.

Connectivity Information Storage:
- Polygon Soup involves not storing any connectivity information, making it a simple representation.
- Shared Vertices (Indexed Face Sets) store both the list of vertices in a face and the other faces that reference the same vertices.
- In the Shared Vertices scheme, each vertex is assigned a unique ID, and faces reference vertices by their IDs.

Polyhedra Representations:
- The text discusses various methods for representing polyhedra structures.
- It mentions two methods, "Polygon Soup" and "Shared Vertex," that were previously discussed.

- A new method called the "Corners" method is introduced, which specifically works for objects composed entirely of triangles.
- The "Corners" method aims to efficiently store connectivity information for triangles.

Corners Method:

- The "Corners" method involves representing triangles through a set of tables.
- The first table is the Geometry Table, which contains the (x, y, z) coordinates of each vertex.
- The second table is the Vertex Table, which contains lists of vertex indices.
- Each group of three vertices in the Vertex Table defines a triangle.
- The order of vertices in each triangle should respect orientation, usually counter-clockwise.
- The third table is the Adjacency Information or "Opposite Table," which provides information about which triangles share an edge.
- In the "Corners" method, each corner corresponds to a triangle vertex but is not the same as a vertex.
- Corners have properties like "next" and "prev," which make it easier to navigate between vertices in the same triangle.
- The "opposite" corner indicates the corner of the adjacent triangle that shares an edge with the current triangle.
- Calculating opposite corners is done by comparing adjacent vertices.

Benefits of the Corners Method:

- The Corners method offers compact storage of triangle meshes.
- It provides efficient ways to find left/right neighbors, triangles sharing an edge, and perform vertex processing.
- This representation is particularly useful for triangle-dominated structures.

Corners Method for Triangle-based Objects

- Introduction:
    - Developed by Jarek Rossingnac, a professor at Georgia Tech.
    - Specifically designed for objects entirely composed of triangles.
- Geometry Table:
    - Represents (x, y, z) coordinates of each vertex.
    - Example:
        - (x1, y1, z1)
        - (x2, y2, z2)
        - ...
- Vertex Table:
    - Lists vertex indices in groups of 3, defining triangles.
    - Example:
        - 0, 1, 2, 1, 0, 3, ...
        - Each group of 3 vertices represents a triangle.
        - Ordering respects orientation (e.g., counter-clockwise).

- Vertex Relationships:
    - For a given corner index "corner," it belongs to triangle # (floor(corner / 3)).
    - The order of vertices in each triangle is crucial for finding adjacent triangles.
- Next and Previous Corners:
    - Define relationships within a triangle.
    - To find the next corner:
        - corner.next = 3 * corner.triangleNum + ((c + 1) % 3)
    - To find the previous corner: corner.next.next.
- Adjacency Information (Opposite Table):
    - Determines which triangles are next to each other.
    - Introduces "corner.opposite" to get the ID of the opposite corner.
- Computing Opposite Corners:
    - Utilizes the Vertex Table.
    - For each corner a and corner b:
        - If (a.next.vertex == b.prev.vertex AND a.prev.vertex == b.next.vertex):
            - opposite[a] = b
            - opposite[b] = a
        - Identifies opposite corners when two triangles share a base.
- Significance of Opposite Corners:
    - Enables easy calculation of left/right neighbors:
        - corner.rightNeighbor = corner.next.opposite
        - corner.leftNeighbor = corner.prev.opposite
    - Facilitates calculating the SWING of a triangle:
        - corner.swing = corner.next.opposite.next
        - Allows rotating to all triangles around a given vertex.

31/10

Convex Hull:

- A convex hull for a set of points is the smallest convex shape that contains all those points.
- It is important for various computational geometry and graphics applications.
- Definition: The smallest convex shape containing a set of points.
- Convex Combination: A point expressed as a "convex" combination of other points in a set.
    - For a line segment p1/p2, a point "a" is a convex combination if: a = w1p1 + w2p2 (w1 + w2 = 1).
    - For a triangle defined by p1/p2/p3, a point "b" is a convex combination if: b = w1p1 + w2p2 + w3*p3 (w1 + w2 + w3 = 1).
    - Generalization: For a set of "n" points p1...pn and a point "c": c = w1p1 + ... + wnpn. If all weights are non-negative and sum up to 1, then c is inside the convex hull.

Convex Combination:

- A convex combination is a point that can be expressed as a combination of other points in a set.
- For a line segment defined by points p1 and p2, a point "a" is a convex combination if it can be written as a weighted sum: a = w1 * p1 + w2 * p2, where w1 + w2 = 1 and w1, w2 ≥ 0.
- For a triangle defined by points p1, p2, and p3, a point "b" is a convex combination if it can be expressed as a weighted sum: b = w1 * p1 + w2 * p2 + w3 * p3, where w1 + w2 + w3 = 1 and w1, w2, w3 ≥ 0.
- Convex combinations are essential for determining if a point lies inside the convex hull of a set of points.

Barycentric Coordinates:

- Barycentric coordinates are used for triangles and represent the convex weights for a given point within the triangle.
- For a point "g" inside a triangle with vertices p1, p2, and p3, the barycentric coordinates are w1, w2, and w3.
- The barycentric coordinates are calculated using the areas of sub-triangles within the main triangle.
- The sum of barycentric coordinates always equals 1 (w1 + w2 + w3 = 1).
- Barycentric coordinates have useful properties in graphics and geometry.
- For triangles, the convex weights w1/w2/w3 for a given point "g" are called barycentric coordinates.
- Computed based on the area of the triangle and its interior sub-sections.
    - W1 = a1/A, W2 = a2/A, W3 = a3/A.
- Barycentric coordinates have useful properties.

Bezier Curves:

- Bezier curves are a family of curves named after Pierre Bezier, and cubic (degree 3) Bezier curves are commonly used.
- Cubic Bezier curves have a start point, an end point, and two intermediate control points.
- Properties of cubic Bezier curves:
- They interpolate through the start and end points.
- The curve is tangent at its endpoints to the line segments joining the control points.
- All points on the curve are inside the convex hull formed by the four control points.
- They provide independent control of endpoints and tangents.
- The equation for a cubic Bezier curve is expressed using basis functions and control points.
- Basis functions, denoted as B1(t), B2(t), B3(t), and B4(t), determine the curve's shape and how it interpolates between control points.
- B1 starts at 1 and sharply drops to 0, B2 and B3 create "humps," and B4 starts at 0 and curves exponentially to 1.
- The curve's shape is defined by weighted combinations of control points based on the basis functions.

- Family of curves named after mathematician Pierre Bezier.
- Focus on Cubic Bezier Curves (degree 3).
- Defined by a start point, an end point, and 2 intermediate "control points."
- Properties:
    - Interpolate through start/end points (p1/p4).
    - Tangent at endpoints to line segments P1/P2 and P3/P4.
    - All points on the curve are inside the convex hull of p1, p2, p3, and p4.
    - Cubic Bezier curves give independent control of endpoints and tangents.
        - Changing one doesn't affect the other, providing flexibility.
    - Suitable for chaining multiple curves into longer curves.
- Parametric Equation of Bezier Curve:
    - Parametric equation for a cubic Bezier curve q(t):

q(t) = B1(t)*P1 + B2(t)*P2 + B3(t)*P3 + B4(t)*P4

        - Where B1 to B4 are basis functions/weights.
    - Basis Functions:
        - B1 starts at 1, then sharply drops off to 0.
        - B2/B3 are "humps" in the middle values of t, tugging the curve towards them.
        - B4 is a flipped version of B1, starting at 0 and curving exponentially to 1.

Bezier Curves and Basis Functions:

- Bezier curves are widely used for creating smooth curves in graphics.
- In cubic Bezier curves, we have four control points: P1, P2, P3, and P4.
- The weights for the Bezier curve are given by four basis functions: B1(t), B2(t), B3(t), and B4(t).

Intuitive Approach to Basis Functions:

- To understand the basis functions, imagine drawing line segments connecting the four control points: P1P2, P2P3, and P3P4.
- Find the midpoints of these line segments: M12, M23, and M34.
- Connect these midpoints to obtain two more line segments: M12M23 and M23M34.
- Continue this process to find additional midpoints, eventually leading to a single point located at Q(1/2), which is the halfway point along the Bezier curve.
- The curve can be divided into two sub-curves: one with control points (P1, L2, L3, L4) and the other with control points (R1, R2, M34, P4).
- This division is performed iteratively until enough points are obtained for drawing a reasonable approximation of the curve.

General Subdivision for Bezier Curves:

- A more efficient way to find points along a Bezier curve is by considering a point t% of the way along each line segment.
- For the P1P2 line segment, the t% point is found using pt = P1 * (1 - t) + P2 * t.
- This approach is used to find points along the curve, starting from a point t% of the way along each line segment.

- The process is repeated to determine the desired point on the curve (e.g., Q(3/4)).

Basis Functions for Cubic Bezier Curves:
- The basis functions for cubic Bezier curves are determined using the subdivision technique.
- Basis functions for the control points:
- B1(t) = (1 - t)^3 for P1
- B2(t) = 3t * (1 - t)^2 for L2 and R1
- B3(t) = 3t^2 * (1 - t) for L3 and R2
- B4(t) = t^3 for P4
- B1(t) and B4(t) are symmetric about t = 1/2 and are inverses of each other.
- Basis functions B2(t) and B3(t) mix the two endpoints.

Significance of Bezier Curves:
- Bezier curves are essential for achieving smooth shapes in graphics.
- Cubic Bezier curves offer independent control of endpoints and tangents, making them popular in computer graphics.
- These curves ensure that shapes maintain their quality at various resolutions, which is crucial for applications like font rendering and architectural drawings.

Polynomial Evaluation and Finite Differences:
- The text introduces polynomial evaluation and finite differences.
- Horner's rule is used to efficiently evaluate polynomial expressions.
- Horner's rule reduces the number of operations by grouping terms.
- Finite differences are used to calculate the differences between function values at equally spaced points, which can simplify evaluation.

Pattern of Numbers:
- The sequence "3, 4, 9, 18, 31" is discussed.
- The pattern involves linear spacing between numbers, resulting in a quadratic appearance.
- The text mentions that the next number in the sequence is 48.

2/11

Complex Numbers and Argand Diagram:
- Complex numbers have both real and imaginary parts, represented as Z = a + b*i.
- Complex numbers can be graphed on a 2D Argand diagram, with the real part (a) on the X-axis and the imaginary part's coefficient (b) on the Y-axis.
- To multiply complex numbers, both terms are multiplied and then separated into real and non-real parts.

Iterating Functions with Complex Numbers:

- When squaring complex numbers (z^2), numbers with a length less than one converge toward the origin, while numbers with a length greater than one diverge.
- Different functions, like f(z) = z^2 + C, yield varying results when iterated, and it's not easy to predict if they'll converge or diverge.
- Iterating these functions creates Julia sets, a type of fractal.
- The initial z-value for the function often comes from screen positions or texture coordinates, with complex coefficients (a, b) based on pixel coordinates (x, y).
- Different values of C lead to different Julia sets, but C remains constant during the iterations.

Drawing Bezier Curves:
- Bezier curves can be drawn using different methods, including solving the cubic basis functions, using the finite difference method, or recursive curve subdivision.
- The finite difference method is the most common approach for drawing Bezier curves.

Introduction to 3D Curves and Surfaces:
- The transition from 2D to 3D curves and surfaces is introduced.
- Parametric surfaces, defined by parameters (s, t), are discussed.
- The analogy is drawn between 2D parameter spaces and 1D parameter spaces for lines.
- For parametric surfaces, three points (P1, P2, P3) are needed to define a 3D parallelogram.

Creating 3D Surfaces:
- The construction of simple 3D surfaces is explained.
- A cylindrical surface is created using parametric equations for x, y, and z.
- A spherical surface is created with latitude (T) and longitude (S) ranges.
- The equations for x, y, and z positions on the sphere are derived.

Intuition Behind Basis Functions:
- Basis functions for a cubic Bezier curve are derived intuitively.
- Visualize line segments connecting control points and midpoints.
- Midpoint at Q(1/2) is essential in determining subdivision points.
- Bezier curve is divided into two curves for approximation.

General Subdivision Technique:
- Point 3/4 along the curve is found by taking 3/4 of the way along each line segment.
- Technique involves continuous subdivision and connection of points to find Q(3/4).

Basis Functions Formulation:
- Basis functions for cubic Bezier curves are crucial for the subdivision.
- Basis functions:
    - For the first control point: B1(t) = (1-t)^3.
    - For the last control point: B4(t) = t^3 (similar to straight line segment weighting).
    - For the 2nd and 3rd control points:
        - B2(t) = 3t*(1-t)^2.
        - B3(t) = 3t^2 * (1-t).

    - B1/B4 and B2/B3 are symmetric about t = 1/2; they're inverses of each other.

Significance of Barycentric Coordinates:

- Barycentric coordinates help in understanding weights for convex combinations.
- Useful in determining a point's position within a convex hull.

Applications of Bezier Curves:

- Bezier curves are crucial for achieving sharp shapes at any resolution.
- Cubic Bezier curves provide independent control of endpoints and tangents.
- Important for applications like font rendering and architectural drawing.

Polynomial Evaluation and Horner's Rule:

- Polynomial evaluation can be optimized using Horner's rule.
- Reduces the number of operations for evaluating a cubic polynomial.
- Formula: f(t) = ((at + b)*t + c)*t + d.

Finite Difference Technique:

- The finite difference technique simplifies computation for higher-order functions.
- Applies to quadratic and higher-order polynomials.
- Example: f(t) = at^2 + bt + c.

Analyzing a Sequence:

- Sequence: "3, 4, 9, 18, 31."
- Recognizing a pattern: Spacing between numbers increases linearly.
- Applying finite difference technique: Understanding the sequence increases quadratically.
- Next number in the sequence: 48.

Complex Numbers and Fractals

Introduction to Complex Numbers:

- Complex numbers have both real (a) and imaginary (b) parts.
- Represented as Z = a + b*i on an Argand diagram (2D plane).

Multiplication of Complex Numbers:

- To multiply complex numbers, distribute over each term.
- Example: Z^2 = a^2 - b^2 + 2abi.

Iterative Function and Julia Sets:

- Iterating the function f(z) = z^2 with different complex numbers results in Julia sets.
- Julia set is a type of fractal.
- Visual representation involves shading based on divergence or convergence.
- Complex coefficients (a, b) based on pixel coordinates (x, y).

Drawing Bezier Curves:

- Methods for drawing 2D Bezier curves:
    - Directly solve cubic basis functions.
    - Use finite difference method for faster computation.
    - Recursive subdivision until acceptable smallness is reached.

Transition to 3D Curves and Parametric Surfaces:

- Introduction to parametric surfaces.

- Consideration of a 2D parameter space (s, t) for defining points on a 3D surface.
- Use of vectors (T and S vectors) for defining a parallelogram in 3D space.

Cylindrical Surface Parametrization:

- Parametrization for a cylindrical surface:
    - Q(s, t) = [x(s, t), y(s, t), z(s, t)].
    - Definitions for x, y, and z:
        - x(s, t) = cos(s).
        - y(s, t) = sin(s).
        - z(s, t) = t.

Spherical Surface Parametrization:

- Parametrization for a spherical surface:
    - Q(s, t) = [x(s, t), y(s, t), z(s, t)].
    - Definitions for x, y, and z:
        - x(s, t) = cos(t) * cos(s).
        - y(s, t) = cos(t) * sin(s).
        - z(s, t) = sin(t).
    - Ranges for T and S are shifted for latitude and longitude representation.

3D Surfaces and Further Exploration:

- Introduction to drawing doubly-curved surfaces like spheres.
- Plans for further exploration into 3D surfaces and parametric representations.

3D Bezier Patches

Introduction to 3D Bezier Patches:

- Extension of Bezier Curves into 3D space.
- 16 control points define a Bezier patch, composed of 4 Bezier curves.
- Commonly used in CAD modeling.

Cubic Bezier Patches:

- Each Bezier curve has 4 control points.
- Parameters S and T define the patch.
- Calculation of 3D points on each curve, creating a new Bezier curve for the surface.
- Explicit process for finding a point Q(s, t) on the Bezier patch.

Surface Through Control Points:

- The patch goes through the 4 corners but not necessarily through other interior points.
- Smooth transition between patches requires matching control points and tangents.

Limitations of Bezier Patches:

- Require a division of the surface into quadrilateral patches.
- Achieving smooth continuity is challenging, needing precise control point matching.
- Mainly used in CAD modeling.

Subdivision Surfaces

Overview of Subdivision Surfaces:

- Modern alternative to Bezier patches in animations and games.

- Provides flexibility with any polygonal mesh.
- Introduced by Catmull-Clark and independently by Catmull-Clark and Dao-Smith in 1978.

Loop Scheme for Subdivision Surfaces:

- Works with triangles.
- Achieves "C2 continuity," allowing two derivatives of the surface to be continuous.
- Adds midpoints to edges and calculates new vertex positions.
- Smoothes the surface by moving old vertices based on valence (number of adjacent triangles).
- Iterative process of subdivision until desired smoothness is achieved.

Loop Scheme Process:

Compute new vertex locations on edges using weighted averages.
Move old vertices to smooth the surface based on valence.
Create smaller triangles by adding midpoints.
Repeat the process iteratively for increased smoothness.

Limitations of Loop Scheme:

- Requires a triangular mesh.
- Has a polygon growth factor of 4.
- Handles extraordinary points with only C1 continuity.

Catmull-Clark Subdivision

Introduction to Catmull-Clark Subdivision:

- Widely used in animation and games, popularized by Pixar.
- C2 continuity almost everywhere.
- Suitable for surfaces with mostly quadrilaterals.

Subdivision Process:

- New vertices added to polygons: center and edges.
- Convex combination for center vertices.
- Smoothing of the surface by moving old vertices.
- C2 continuity achieved for most points.

Sharp Edges in Catmull-Clark:

- Identification of sharp edges by tagging.
- Formulation for creating new points on sharp edges.
- Adjustment for old vertices between two sharp edges.

Semi-Sharp Edges:

- Introduction of a sharpness parameter, "S."
- Application of sharp subdivision rules "S" times.
- Subsequent smoothing to maintain edge shape.

Limitations:

- Extraordinary vertices exhibit C1 continuity instead of C2.
- Sharpness parameter for semi-sharp edges simplifies the process.

Regular Polygons and Platonic Solids

Introduction to Regular Polygons:

- Convex polygons with equal side lengths and angles.
- Examples include equilateral triangles, squares, and regular polygons up to octagons.
- Heptagon is often elusive, but real-life encounters happen unexpectedly.

Regular Planar Tiling and Platonic Solids

Regular Planar Tiling:

- Challenge: Fill an infinite plane with one type of regular polygon, same size, no gaps or overlaps.
- Square tiling is straightforward with squares having 4 sides and valence 4.
- Dual tiling swaps vertices and faces, resulting in self-dual square and valence 4 vertices.
- Triangular tiling creates hexagonal shapes with 3 sides and valence 6, dual to hexagonal tiling.
- Hexagonal tiling (Honeycomb tiling) with 6 sides and valence 3, dual to triangular tiling.

Platonic Solids:

Cube/Octahedron Duality:

- A cube has 6 faces, 8 vertices, and 12 edges.
- Dual tiling operation results in an octahedron with 8 faces, 6 vertices, and 12 edges.
- Cube and octahedron are duals of each other.

Tetrahedron:

- Tetrahedron has 4 faces, 4 vertices, and 6 edges.
- Dual tiling creates another tetrahedron; it is self-dual.

Icosahedron/Dodecahedron Duality:

- An icosahedron has 20 faces, 12 vertices, and 30 edges.
- Dual tiling operation results in a dodecahedron with 12 faces, 20 vertices, and 30 edges.
- Icosahedron and dodecahedron are duals of each other.

Applications of Platonic Solids:

- Platonic solids are used for various purposes:
    - Dice with many sides.
    - Soccer balls resemble truncated icosahedra.
    - Radiolarans exhibit an icosahedral body.
    - Geodesic domes utilize the icosahedron's subdivision into a sphere.
    - Cubes and octets of tetrahedrons/octahedrons tile 3D spaces.
    - Platonic solid-shaped lattices in chemistry.
    - Starting blocks for building shapes in computer graphics.

Euler's Equation:

- $V + F = E + 2$
- Euler's equation holds for platonic solids and shapes without "handles."
- "+2" accounts for the number of volumes, representing inside and outside for 3D solids.
- Generalizes to any 3D solid without handles, emphasizing the relationship between even and odd-dimensional geometric objects.

Fractals: Infinite Detail and Dimensionality

Introduction:

- Fractals are infinitely detailed geometric objects prevalent in nature, displaying recursion and iteration on geometry.
- Georg Cantor's pattern involves removing the middle third of a line segment infinitely, resulting in a fractal dust with infinite unconnected points but zero total length.
- Sierpinski's Carpet and Peano Curves are 2D fractal examples that fill surfaces with holes yet remain a single connected surface.
- The Koch Snowflake, formed by iteratively creating triangles on line segments, has infinite length but no tangent anywhere.

Dimensionality in Geometry:

- Dimensionality defines how a shape's size changes with scale; Hausdorff-Besicovitch dimension relates repetition and linear scale factors.
- A 2D square's dimension is log(4)/log(2) = 2, while the Koch Snowflake's dimension is log(4)/log(3) ≈ 1.2, showcasing non-integer dimensions.

Applications of Fractals:

- Mandelbrot and Julia sets are famous fractals with applications in diverse fields:
    - Blood vessels, rivers, and star distributions in nature exhibit fractal-like patterns.
    - Carpenter's midpoint algorithm generates realistic 2D and 3D mountain shapes.
    - Procedural landscapes in film and games often use algorithms derived from fractals.
    - Julia sets are analyzed based on whether the corresponding point in the Mandelbrot set is connected or disconnected.
    - Mandelbrot sets, formed by varying C while keeping Z fixed, are connected wholes.

Higher-Dimensional Fractals:

- Mandelbulb is a 4D variant of the Mandelbrot set, displaying intricate structures.
- Fractals, initially mathematical oddities, have practical applications and are extensively observed in the natural world.

Conclusion:

- The study of fractals, combining mathematical abstraction with real-world observations, reveals the ubiquity of these patterns and their impact on various fields.
- Fractals provide a unique lens for understanding nature's complexity and have become integral in applications ranging from computer graphics to the modeling of natural phenomena.

Volume Rendering: Unveiling the Depths

Introduction:

- Volume rendering, a crucial technique in the medical field, involves representing 3D objects through volume data, organized in a 3D grid with each point referred to as a voxel.

- The data sources for volume rendering span various domains, including medical imaging (CAT scans, MRIs), engineering datasets (fluid simulations, stress/strain information), and more.

Rendering Approaches:

- Two primary approaches for rendering voxels are Iso-Surface methods and Direct methods.
    - Iso-Surface Methods: Involve creating polygons based on voxel data.
    - Direct Methods: Generate a rendering directly from voxel data using techniques like raytracing, often suitable for scenarios requiring transparency.

Iso-Surface Rendering: Marching Cubes Algorithm:

- To improve over a basic cube representation, sophisticated techniques like the Marching Cubes algorithm are employed.
- An illustrative introduction to a simpler version, "Marching Squares," is provided for clarity.
    - Consider a 2D grid with scalar values in each voxel, and a cutoff value (e.g., 5) distinguishing "inside" and "outside" of a shape.
    - Label voxels as inside/outside, and along line segments connecting voxels, linearly interpolate the cutoff value between inside and outside points.
    - Connect these interpolated points to form the shape; in 3D, polygons connect these points.

Conclusion:

- Volume rendering, integral in medical imaging and beyond, employs sophisticated algorithms to translate volumetric data into meaningful visualizations.
- Iso-Surface methods, exemplified by the Marching Cubes algorithm, enhance rendering by creating intricate representations from voxel data.
- Direct methods, while useful for transparency needs, underscore the versatility of volume rendering techniques in diverse applications.

Volume Rendering: Illuminating 3D Worlds

Marching Cubes Algorithm:

- The Marching Cubes algorithm, prominent in computer graphics, facilitates rendering 3D models stored as voxel arrays.
- It considers a neighborhood of 8 voxels forming a cube, distinguishing "inside" and "outside" to create polygonal surfaces.

Direct Rendering Approach:

- Direct rendering involves classifying voxels, calculating shading, and rendering based on voxel data.
- A common technique, raytracing, follows a three-stage process: classification, shading, and actual rendering.

Raytracing for Voxels:

- Raytracing considers opacity and color within voxels, assuming each voxel is filled with identical, opaque particles.

- Opacity and color calculation involves tri-linear interpolation and classification based on density values from data sources like CAT scans.
    - Opacity categorization might involve thresholds (e.g., 0 <= V <= 0.2 is dark green, 0.2 <= V <= 0.5 is regular green, etc.).

Raytracing for Voxels:

- Raytracing considers opacity and color within voxels, assuming each voxel is filled with identical, opaque particles.
- Opacity and color calculation involves tri-linear interpolation and classification based on density values from data sources like CAT scans.

Volume Rendering Wrap-up:

- Interpolating within voxels accommodates varying materials' transparency.
- Opacity determination is derived from density values associated with different materials in data, with certain real-world challenges acknowledged.

Transition to Non-Photorealistic Rendering:

- Non-photorealistic rendering, aimed at capturing hand-drawn aesthetics, introduces cel shading.
- Cel shading employs dark outlines, achieved through techniques like edge detection, and simplifies shading with discrete color categories.

Wind Waker's Cel Shading:

- "The Legend of Zelda: The Wind Waker" stands out as an early example of real-time cel shading in video games.
- Cel shading imparts a distinct cartoony appearance, using dark outlines and simplified shading, departing from traditional 3D rendering styles.

Fluid Simulation Applications:

- Animated movies
- Computer games
- Medical simulations (e.g., blood flow in the heart)
- Enjoyment and entertainment

Computational Fluid Dynamics (CFD):

- Utilizes engineering and mathematical simulations.
- Graphics applications often favor the "finite differences" method.

Navier-Stokes Equations:

- Two equations:
    - Incompressibility Equation: laplacian of u equals 0.
    - Velocity Change Equation: u_t equals k times the laplacian of u minus u dot gradient times u minus nabla p plus f.

Finite Difference Grids:

- Regular square grids for scalar and vector fields.
- Scalar field represents material quantity.
- Vector field represents fluid velocities.

Diffusion Term:

- Equation: c_t equals k times the laplacian of c_t.
- Diffuses material based on neighboring values.
- Equivalent to Gaussian blur mathematically.

Diffusion of Velocity:

- Represented as u_ij equals (xVel equals u_ij^x, yVel equals u_ij^y).
- Diffusion of x/y velocities independently: u_ij^x equals k times the laplacian of u_ij^x and u_ij^y equals k times the laplacian of u_ij^y.

Advection Term:

- Equation: u_t equals minus u dot gradient times u.
- Semi-Lagrangian advection moves material to chosen destinations.

Pressure Term and Divergence:

- Divergence measures material flow.
- Enforce incompressibility through pressure calculation.

Pressure Projection:

- Adjust velocity and pressure to achieve divergence-free vector field.

Fluid Simulation Steps:

1. Diffuse velocities.
2. Advect velocities.
3. Add body forces (e.g., gravity).
4. Pressure projection (calculate pressures).
5. Diffuse scalar values (amount of material).
6. Advect scalar values.
7. Repeat from step #1.

Modeling Objects Inside Fluid:

- Use rigid-body simulations similar to pressure projection calculations.

Handling Different Fluid Sizes:

- Surface tension becomes crucial at small scales (centimeter-sizes).
- Hydrophilic surfaces cause spreading; hydrophobic surfaces lead to spherical beads.

Key Components of VR Systems:

- Track user's head motion.
- Create real-time images of the virtual world.
- Display those images convincingly to the user.

History of VR:

- 1960s: Ivan Sutherland explored VR fundamentals, created the "Ultimate Display."
- Late 1980s: Jaron Lanier founded VPL, introduced term "virtual reality."
- 2010: Palmer Luckey created Oculus VR prototype, improved motion tracking.
- 2014: Facebook acquired Oculus VR for ~3 billion dollars.

Tech in Modern VR Systems:

- GPUs for rendering millions of polygons at high frame rates.
- Head-Mounted Display (HMD) technology:
    - Separate images for each eye.

        - Typically uses LCD or OLED displays with lenses.
        - Wide field-of-view, lightweight, wireless for user comfort.

Oculus Rift's Headset:

- Uses Fresnel lenses for a wide field-of-view.
- Slices in the lens provide a smaller, lighter design.

Tracking Systems in VR:

- Concerns: Determine x, y, z position, and head orientation at 60 Hz or faster.
- Techniques:
    - Magnetic tracking systems: Fallen out of favor due to metal interference.
    - Optical systems:
        - Outside-in: External camera tracks beacons on the HMD.
        - Inside-out: HMD's camera observes room reference points.

Challenges for VR Today:

- Motion sickness remains a problem.
- Lack of a "killer app" for widespread adoption.
- Augmented reality (AR) competition from Magic Leap and Hololens.
- Uncertain if the current wave of VR will become mainstream.

Introduction to Game Engines:

- Games involve various components (collision detection, physics, animation, sound), with a focus on game rendering techniques.
- Real-time rendering aims to balance high-quality images with fast render speeds.
- Popular game engines include Unity, Unreal, Cryengine, and Source, often free for educational use.

Rendering in Games:

- Two rendering types:
    - Immediate-mode rendering: CPU sends polygons to GPU one-by-one.
    - Retained-mode rendering: Polygons sent to GPU once, stored in vertex buffer object (VBO).

Efficiency Strategies:

- Draw fewer polygons to increase speed.
- Techniques like Potentially Visible Sets, Portals, and Level-of-Detail meshes.
- Tessellation for adding polygons based on proximity, often combined with Displacement Maps.

Lighting Techniques:

- Burned-in lighting for pre-computed lighting stored in lightmap textures.
- Ambient Occlusion for estimating indirect light, enhancing shading.
- Challenges include reflective surfaces and moving objects.

Post-Processing in Games:

- Pixel shaders modify images after rendering.
- Techniques include motion blur, depth-of-field, vignettes, fog/haze, light shafts, bloom, and lens flares.

Real-Time Ray Tracing:

- Recent advancement in games for realistic effects.
- Ray tracing benefits include global illumination, correct reflections, depth-of-field, ambient occlusion, and soft shadows.
- Often used as mixed effects alongside rasterization for most scenes.

Challenges and Future Trends:

- Real-time ray tracing still evolving, often used in combination with traditional rendering.
- Continuous efforts for efficiency improvements and realistic effects in game engines.

engines: Unity, Unreal,
Cryengine, Source, etc. (many of which are free for educational use -
they wanna get you hooked as a student)

Procedural Content Generation (PCG) Overview:

- PCG is a technique used in games and other applications to generate content using algorithms, reducing the manual creation burden.
- The primary goal is to create a variety of objects with different shapes, sizes, colors, and features, often relying on random numbers.

Applications of PCG:

- Widely used in games like Minecraft for terrain generation, Simcity for placing trees and NPCs, and Spore for generating creatures.

Perlin Noise:

- Perlin Noise is a famous technique for generating "controlled random numbers" in the range [0, 1].
- Useful for creating textures, hills, and patterns where related values are needed.
- Applied in various ways, such as bump maps, turbulence for different patterns, and more.

Terrain Generation with PCG:

- PCG is commonly used to create terrain for games and movies, involving heightmaps and fractals.
- Simulation of erosion by water and wind, considering grid-based rain deposition to sculpt valleys and rivulets.
- Research explores user-guided terrain generation for added control.

Plant Generation using Grammars:

- Grammars, like L-Systems, are used for plant generation, providing rules for branching and growth.
- Randomness introduced through assigning probabilities to rules or outcomes (e.g., petal count in flowers).

Building Generation with Grammars:

- Similar grammar systems used for building objects, allowing rules to add windows, floors, and modify floorplans.

- Examples include generating city plans for ancient Roman towns, applying grammar rules for street layout, and filling blocks with building models.

City Generation in Games:

- Games like "Cities: Skylines" use PCG for city layouts, controlling traffic patterns, individual buildings, and more.
- Artists can provide input for road directions, building placements, etc., using algorithms for the overall design.