6. Theory of Computation and Computer Graphics

6.1 Introduction to Finite Automata

- Introduction to Finite Automata and Finite State Machines
- Equivalence of DFA and NDFA
- Minimization of Finite State Machines
- Regular Expressions
- Equivalence of Regular Expression and Finite Automata
- Pumping Lemma for Regular Language

6.2 Introduction to Context-Free Language

- Introduction to Context-Free Grammar (CFG)
- Derivative Trees (Bottom-up and Top-down Approach, Leftmost and Rightmost Derivation)
- Language of a Grammar
- Parse Tree and its Construction
- Ambiguous Grammar
- Chomsky Normal Form (CNF), Greibach Normal Form (GNF), Backus-Naur Form (BNF)
- Push Down Automata
- Equivalence of Context-Free Language and PDA
- Pumping Lemma for Context-Free Language
- Properties of Context-Free Language

6.3 Turing Machine

- Introduction to Turing Machines (TM)
- Notations of Turing Machine
- Acceptance of a String by a Turing Machine
- Turing Machine as a Language Recognizer
- Turing Machine as a Computing Function
- Turing Machine as an Enumerator of Strings of a Language
- Turing Machine with Multiple Tracks
- Turing Machine with Multiple Tapes
- Non-Deterministic Turing Machines
- Church-Turing Thesis
- Universal Turing Machine for Encoding of Turing Machines
- Computational Complexity, Time and Space Complexity of a Turing Machine
- Intractability, Reducibility

6.4 Introduction to Computer Graphics

- Overview of Computer Graphics
- Graphics Hardware (Display Technology, Architecture of Raster-Scan Displays, Vector Displays, Display Processors, Output and Input Devices)
- Graphics Software and Software Standards

6.5 Two-Dimensional Transformation

- Two-Dimensional Translation, Rotation, Scaling, Reflection, Shear Transformation
- 2D Composite Transformation
- 2D Viewing Pipeline

- World to Screen Viewing Transformation and Clipping (Cohen-Sutherland Line Clipping, Liang-Barsky Line Clipping)

6.6 Three-Dimensional Transformation

- Three-Dimensional Translation, Rotation, Scaling, Reflection, Shear Transformation
- 3D Composite Transformation
- 3D Viewing Pipeline
- Projection Concepts (Orthographic, Parallel, Perspective Projection)

6.1 Introduction to Finite Automata

Finite Automata (FA) are a type of **mathematical model** used to represent and analyze regular languages. These are simple computational models that can be in one of a finite number of states at any given time. A finite automaton transitions between states based on inputs, and the automaton can either accept or reject a string based on whether it ends in an accepting state.

---

1. Introduction to Finite Automata and Finite State Machine (FSM)

Finite Automata are mathematical models of computation used to design and analyze systems like parsers, compilers, and text search engines. They are used for recognizing patterns in strings and are foundational concepts in the theory of computation and regular languages. It consists of:

- A finite set of **states** (one of which is the initial state, and some of which are accepting states).
- A **set of input symbols** (often called an alphabet).
- A **transition function** that defines how the machine moves between states based on input symbols.
- An **initial state** (where the machine starts).
- A set of **accepting states** (where the machine accepts the input string if it ends in one of these states).

Finite State Machine (FSM) is essentially another term for **Finite Automata**, which can be of two types:

- **Deterministic Finite Automaton (DFA)**: Every state has exactly one transition for each input symbol.
- **Nondeterministic Finite Automaton (NDFA or NFA)**: A state can have multiple transitions for the same input symbol, or even none at all.

DFA and **NFA** are both equivalent in terms of the languages they can recognize, i.e., both can recognize **regular languages**.

---

2. Equivalence of DFA and NDFA

1. **DFA (Deterministic Finite Automaton)**

A **Deterministic Finite Automaton (DFA)** is a finite state machine where:

- **Deterministic**: For each state and input symbol, there is exactly one transition to a next state.

- The machine can only be in one state at a time.
- Once the input is completely processed, the DFA decides if the input string belongs to the language (accepts or rejects it).

Components of DFA

A DFA consists of the following components:

- **Q**: Finite set of states.
- **Σ**: Input alphabet (set of allowed input symbols).
- **δ**: Transition function ($\delta: Q \times \Sigma \to Q$) that maps a state and an input symbol to a single next state.
- $\mathbf{q_0}$: Initial state ($q_0 \in Q$).
- **F**: Set of accepting (final) states ($F \subseteq Q$).

Example

For a language **L** = {w ∈ {a, b}* : w ends with “ab”}, a DFA would:

- Transition between states based on the last two characters of the string.
- Accept the string if it ends in “ab”.

Applications of DFA

DFA is widely used in various domains, including:

- **Pattern matching** in text editors (e.g., searching for a specific word).
- **Lexical analysis** in compilers.
- Traffic lights operate based on a deterministic finite set of states (e.g., Red → Green → Yellow → Red).

---

2. **NFA (Nondeterministic Finite Automaton)**

A **Nondeterministic Finite Automaton (NFA)** is a finite state machine where:

- **Nondeterministic**: For a given state and input symbol, there may be multiple transitions or no transition at all.
- The machine can be in multiple states simultaneously.
- NFAs allow **ε-transitions**, which enable transitions without consuming any input symbol.

Components of an NFA

An NFA consists of the following components:

- **Q**: Finite set of states.
- **Σ**: Input alphabet.
- **δ**: Transition function ($\delta: Q \times \Sigma \to 2^{P}(Q)$), which maps a state and an input symbol to a set of possible next states.
- $\mathbf{q_0}$: Initial state.
- **F**: Set of accepting (final) states.

Example

For the same language **L** = {w ∈ {a, b}* : w ends with “ab”}, an NFA would:

- Allow multiple paths to traverse the states simultaneously.
- Accept a string if at least one path leads to a final state.

Applications of NFA

NFAs are used in various domains, including:

- **Modeling real-life systems** with uncertainty (e.g., speech recognition).
- Used as a **theoretical concept** in automata theory to simplify proofs and design algorithms.
- Serve as the **basis for converting into DFA** for practical implementation in lexical analyzers.
- NFAs are used to match words with possible typos by exploring multiple paths simultaneously.

Key Differences Between DFA and NFA

Aspect	DFA	NFA
Determinism	One unique transition per input symbol from each state.	Multiple transitions or ε-transitions are allowed.
State	Only one state active at a time.	Can be in multiple states simultaneously.
Ease of Implementation	Easier to implement due to deterministic nature.	Harder to implement directly in software or hardware.
Number of States	Often requires more states.	Can have fewer states for the same language.
Processing Speed	Faster as only one path is explored.	Slower due to multiple simultaneous paths.
Power	Equally powerful (both recognize regular languages).	Equally powerful (both recognize regular languages).

Although DFA and NFA differ in their structures and transition rules, **both are equivalent in power**, meaning they can recognize the same class of languages: **regular languages**.

NFA to DFA Conversion (Subset Construction Algorithm):

1. Start with the initial state of the NFA, which becomes the initial state of the DFA.
2. Create new states in the DFA by taking all possible combinations (subsets) of states in the NFA.
3. For each subset, determine the transition to other subsets based on input symbols.
4. Mark a subset as accepting if it contains at least one accepting state of the NFA.

   Challenge

   This conversion can lead to an **exponential increase** in the number of states. For example, an NFA with **n** states can result in a DFA with up to $\mathbf{2^n}$ states.

DFA to NFA Conversion:

Since a DFA is a special case of an NFA (where each state has exactly one transition per input symbol), a DFA can trivially be treated as an NFA.

---

3. Minimization of Finite State Machines

The **minimization** of finite state machines (FSMs) is the process of reducing the number of states in a DFA without changing the language it recognizes. This is important for optimizing the machine and making it more efficient.

The minimization process typically involves:

1. **Removing unreachable states**: States that are not reachable from the initial state are removed.
2. **Merging equivalent states**: Two states are considered equivalent if they behave in the same way for all possible inputs. These states can be merged into one.

---

4. Regular Expressions

A **regular expression** is a formal notation used to describe patterns in strings. Regular expressions can be used to define the syntax of regular languages and can be converted into a finite automaton.

Key components of regular expressions:

- **Symbols**: Basic characters in the alphabet.
- **Concatenation**: Placing two regular expressions together (e.g., `ab`).
- **Union (alternation)**: Choosing between two expressions (e.g., `a|b`).
- **Kleene star**: Repeating the expression zero or more times (e.g., `a*`).
- **Parentheses**: Grouping expressions to enforce precedence (e.g., `(ab|cd)*`).

Regular expressions provide a compact way to represent regular languages and can be directly converted into **finite automata** (both DFA and NFA).

---

5. Equivalence of Regular Expression and Finite Automata

Regular expressions and **finite automata** are two different ways to describe the same set of languages: the **regular languages**. A **regular language** can be described either by a **finite automaton** (DFA or NFA), or a **regular expression**.

These two models are equivalent in terms of the languages they recognize. Any **regular expression** can be converted into a **finite automaton**, and vice versa.

- **From Regular Expression to Finite Automaton**: Using algorithms like **Thompson's construction**, you can convert a regular expression into an NFA. After that, it can be converted into a DFA if needed.
- **From Finite Automaton to Regular Expression**: You can derive a regular expression from a finite automaton by using methods like state elimination or algebraic approaches.

---

6. Pumping Lemma for Regular Language

The Pumping Lemma is a property of regular languages that can be used to prove that a given language is not regular. It states that:

If a language `L` is regular, then there exists a pumping length `p` such that any string `s` in `L` with length at least `p` can be divided into three parts, `s = xyz`, such that:

- `|xy| ≤ p` (the length of `xy` is at most `p`),
- `|y| > 0` (the string `y` is non-empty),
- `xy^n z ∈ L` for all `n ≥ 0` (repeating the `y`-part any number of times will result in a string that is still in the language).

This lemma is used to show that certain languages do not have the properties required to be regular languages. If you cannot find such a decomposition of a string, then the language is not regular.

Example:

Consider the language `L = {a^n b^n | n ≥ 0}`. Using the pumping lemma, you can prove that this language is not regular.

Conclusion

- **Finite Automata** (DFA and NFA) are theoretical models for recognizing regular languages. They operate by transitioning through states based on input symbols.
- **DFA and NFA** are equivalent in the languages they recognize, although DFAs are deterministic and NFAs allow multiple or no transitions for the same input.
- **Minimization of FSMs** reduces the number of states in a DFA while maintaining the language it accepts.
- **Regular Expressions** provide a compact way to describe regular languages and are equivalent to finite automata in terms of language recognition.
- The **Pumping Lemma** is used to prove that certain languages are not regular by showing that they cannot be "pumped" in the way required by regular languages.

6.2 Introduction to Context-Free Languages (CFL)

In this section, we will dive into **Context-Free Languages (CFLs)**, which are a broad class of formal languages that can be generated by a specific type of grammar called **Context-Free Grammar (CFG)**. CFLs are widely used to model the syntax of programming languages and are central to the study of compilers and language processing.

---

1. Context-Free Grammar (CFG)

A **Context-Free Grammar (CFG)** is a formal grammar in which each production rule has a **single non-terminal symbol** on the left-hand side. CFGs are used to define the syntax of context-free languages. A CFG consists of:

- A set of **non-terminal symbols** (used to define the language's structure).
- A set of **terminal symbols** (the basic symbols of the language, which appear in strings).
- A set of **production rules** (or rewrite rules) that describe how non-terminal symbols can be replaced by other symbols.
- A **start symbol** (the symbol from which the derivation starts).

The production rules are of the form:

```
Non-terminal → String of terminals and/or non-terminals
```

For example:

```
S → aSb | ε
```

This defines a language where **S** can be replaced by `aSb` or the empty string `ε`. This generates strings with balanced numbers of `a`'s and `b`'s, such as `ab`, `aabb`, `aaabbb`, etc.

---

2. Derivative Trees (Top-down and Bottom-up approach, Leftmost and Rightmost Derivation)

A **derivation tree** (or parse tree) visually represents how a string in a language can be derived using a grammar's production rules. It provides a structured way to illustrate the step-by-step process of deriving a string from the start symbol.

Derivation trees are central to **parsing** in compilers, where they help in analyzing the syntax of programming languages. They can be constructed using two main approaches:

---

1. **Top-down Approach**

In this approach, the derivation starts from the **start symbol** of the grammar and breaks it down into the terminals step by step using production rules. The process continues until the string (sequence of terminals) is completely derived.

Example

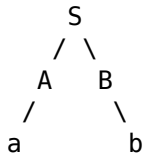
Consider the grammar:

- **S → AB**
- **A → a**
- **B → b**

To derive the string **ab**:

1. Start with **S**.
2. Apply **S → AB**.
3. Replace **A** with **a** using **A → a**.
4. Replace **B** with **b** using **B → b**.

The derivation tree will look like:



---

2. **Bottom-up Approach**

In this approach, the derivation starts with the **terminal string** (sequence of terminals) and reduces it to the **start symbol** step by step. This is the reverse of the top-down approach and is often used in **shift-reduce parsing**.

Example

For the same grammar, derive **S** from **ab**:

1. Start with **ab**.
2. Combine **b** into **B** using **B → b**.
3. Combine **a** into **A** using **A → a**.
4. Combine **AB** into **S** using **S → AB**.

The reduction steps:

```
Step 1: ab
Step 2: aB   (using B → b)
Step 3: AB   (using A → a)
Step 4: S    (using S → AB)
```

---

3. **Leftmost Derivation**

At each step, the **leftmost non-terminal** in the current string is replaced using one of its production rules. It is widely used in **recursive-descent parsers**.

Example

For the grammar:

- **S → AB**
- **A → a**
- **B → b**

To derive **ab** using **leftmost derivation**:

1. Start with **S**.
2. Replace the leftmost **S** with **AB**: **AB**.
3. Replace the leftmost **A** with **a**: **aB**.
4. Replace the leftmost **B** with **b**: **ab**.

Steps:

```
S ⇒ AB ⇒ aB ⇒ ab
```

---

4. **Rightmost Derivation**

At each step, the **rightmost non-terminal** in the current string is replaced using one of its production rules. It is used in **LR parsers** (common in compiler design).

Example

Using the same grammar, derive **ab** with **rightmost derivation**:

1. Start with **S**.
2. Replace the rightmost **S** with **AB**: **AB**.
3. Replace the rightmost **B** with **b**: **Ab**.
4. Replace the rightmost **A** with **a**: **ab**.

Steps:

```
S ⇒ AB ⇒ Ab ⇒ ab
```

Key Differences Between Leftmost and Rightmost Derivations

Aspect	**Leftmost Derivation**	**Rightmost Derivation**
Replaced Non-terminal	Leftmost non-terminal first	Rightmost non-terminal first
Order of Parsing	Natural for recursive-descent parsers	Used in bottom-up parsers (like LR)
Output Order	Produces parse tree top-down	Produces parse tree bottom-up

---

Applications of Derivation Trees

1. **Parsing in Compilers**:

- Helps validate the syntactical structure of a programming language.

2. **Grammar Analysis**:
   - Derivation trees illustrate ambiguities in grammars.
3. **Understanding Language Rules**:
   - Derivation trees are useful in formal language theory to understand the generative process of a grammar.
4. **Compiler Design**:
   - Used in constructing parsers, such as LL parsers (top-down) and LR parsers (bottom-up).
5. **Syntax Trees in Programming**:
   - Similar structures are used for abstract syntax trees in programming languages.

---

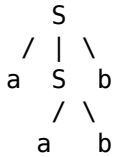
3. Parse Tree and Its Construction

A **parse tree** is a tree that represents the syntactic structure of a string according to a given grammar. It is a **visual representation** of the derivation process, where:

- The **root** of the tree is the start symbol.
- **Internal nodes** represent non-terminal symbols.
- **Leaves** represent terminal symbols (actual characters from the string).

The parse tree is constructed by recursively applying the grammar rules starting from the start symbol, ensuring that the string is derived according to the grammar.

Example of a parse tree for the string `aabb` using the grammar `S → aSb | ε`:



---

4. Ambigous Grammar

A grammar is **ambiguous** if there exists a string that can be generated by the grammar in more than one way (i.e., it has more than one valid parse tree). Ambiguity in a grammar can be problematic, especially when designing compilers, as it can lead to multiple interpretations of the same string.

For example, the following grammar for arithmetic expressions:

`E → E + E | E * E | (E) | id`

This grammar describes how an expression `E` can be built. It allows:

- An expression to be the sum of two expressions (`E + E`).
- An expression to be the product of two expressions (`E * E`).
- An expression to be enclosed in parentheses (`(E)`).
- An expression to be a simple identifier (`id`).

The string `id + id * id` has more than one valid parse tree, leading to ambiguity in how the expression should be evaluated.

Why This is Ambiguous?

The same string, `id + id * id`, can be parsed in two different ways:

1. **Addition first**, followed by multiplication.
2. **Multiplication first**, followed by addition.

This leads to ambiguity because there is no way to tell from the grammar itself whether the multiplication or the addition should be performed first.

Problems of Ambiguity

1. **Multiple Interpretations**: Ambiguity causes multiple valid parse trees, leading to different interpretations of the same input string. For arithmetic expressions, the most common ambiguity arises from **operator precedence**—should multiplication or addition take precedence? In most programming languages, multiplication has higher precedence than addition, but the given grammar doesn't specify this, leading to ambiguity.
2. **Compilers**: When a compiler or interpreter encounters an ambiguous grammar, it may not know how to correctly parse the input. This could lead to errors or unexpected results in the output program. Compilers need **unambiguous grammars** to ensure they can generate the correct machine code or interpret the program as intended.
3. **Parsing Performance**: Ambiguity in a grammar makes parsing more complex, as the parser may need to explore multiple parse trees and choose the correct one based on further context or rules, increasing computational overhead.

Ambiguity in grammars is problematic in formal languages and programming languages because it leads to multiple interpretations of the same input. Resolving this ambiguity usually involves restructuring the grammar to reflect operator precedence and associativity rules. This is especially important when designing compilers or interpreters that need to generate or interpret code accurately.

---

5. Chomsky Normal Form (CNF)

Chomsky Normal Form (CNF) is a standardized form of context-free grammar (CFG) that simplifies certain operations, such as parsing and proving properties of context-free languages. CNF restricts the production rules of a CFG to a specific structure, making it easier to analyze and work with. The CNF is essential for certain algorithms, especially parsing algorithms like the CYK (Cocke-Younger-Kasami) algorithm.

1. **Definition of Chomsky Normal Form**

A grammar is said to be in Chomsky Normal Form (CNF) if every production rule adheres to one of the following three forms:

- **$A \rightarrow BC$**

  Where A, B, and C are non-terminal symbols. This means that a non-terminal can only produce two other non-terminals. This is referred to as binary branching.

- **$A \rightarrow a$**

  Where A is a non-terminal and a is a terminal symbol. In this case, a non-terminal produces a single terminal symbol.

- **$A \rightarrow \varepsilon$**

  This rule can only occur for the start symbol, and only if the start symbol can derive the empty string (ε). The empty string rule is a special case and is allowed only for the start symbol in CNF.

---

2. **Restrictions in CNF**

In CNF, there are a few important restrictions on the form of production rules:

- **No direct terminal to non-terminal productions:**

  In a standard context-free grammar, a production might allow a non-terminal to be replaced by a string of terminals, such as:

  `A → aB`

  where `a` is a terminal symbol, and `B` is a non-terminal. In CNF, this would not be allowed. Instead, terminal symbols must be produced by a single non-terminal, i.e., `A → a`, and non-terminals must always be replaced by either two non-terminals or an empty string in the case of the start symbol.

- **Binary branching only:**

  Productions must be binary. This means that a non-terminal must either produce two non-terminals or one terminal. There can be no production where a non-terminal generates more than two symbols, like `A → BCD`, or fewer than two symbols, like `A → B`.

---

3. **Why Use CNF?**

The structure of CNF is useful for several reasons:

- **Efficient Parsing:**

  CNF is useful for parsing algorithms such as the CYK algorithm, which operates efficiently on grammars in CNF. These algorithms rely on the fact that every rule has at most two non-terminals on the right-hand side, simplifying the process of constructing parse trees.

- **Theoretical Simplicity:**

  CNF simplifies the theoretical analysis of context-free languages. It helps in proving properties like pumping lemmas and the existence of a parse tree for any derivable string.

- **Uniformity:**

  CNF restricts all production rules to a consistent and uniform form, making the grammar easier to work with in formal proofs and computational models.

---

4. **Conversion to CNF**

Converting a general context-free grammar (CFG) to CNF is a process that involves several steps. Here's an overview of the steps involved in converting a CFG to CNF:

- **Remove ε-productions (Empty String Productions)**

If the start symbol can derive the empty string ε, we need to create new productions that account for this possibility. For each rule where a non-terminal can derive the empty string, we need to add alternative productions that account for the presence or absence of that non-terminal.

For example, if we have a rule like:

`A → ε`

we may need to modify rules where A appears and generate alternative versions where A is absent.

- **Eliminate Unit Productions**

A unit production is a rule where a non-terminal produces exactly one non-terminal:

`A → B`

These productions must be eliminated by substituting B's production rules into the rule for A.

- **Eliminate Terminal Symbols from Right-hand Sides of Productions (Other Than Single Terminal Productions)**

If a rule has a terminal symbol combined with non-terminals, such as:

`A → aB`

We need to replace `a` (the terminal) with a new non-terminal `X` that produces `a` (i.e., `X → a`). Then, we substitute this new non-terminal into the rule:

`A → XB`

- **Convert Long Productions to Binary Form**

If a production has more than two symbols on the right-hand side, such as:

`A → BCD`

We break it down into binary form. First, introduce a new non-terminal `X`:

```
A → BX
X → CD
```

This transformation ensures that the grammar follows the binary branching rule.

Chomsky Normal Form is a strict form of context-free grammar that simplifies the process of parsing and analyzing context-free languages. By restricting production rules to binary branching or single terminal symbols, CNF makes certain algorithms more efficient and easier to implement. While many real-world grammars may not be in CNF, converting them into CNF is often necessary for certain types of parsing algorithms and theoretical analysis.

---

6. Greibach Normal Form (GNF)

A **Greibach Normal Form (GNF)** is another form of context-free grammar where every production rule has the form:

`A → aα`

Where `a` is a terminal symbol, and `α` is a (possibly empty) string of non-terminals. This form ensures that every production starts with a terminal symbol.

---

7. Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a notation used to express the **syntax of programming languages** and formal grammars. It is a way of describing the rules for forming strings in a language. BNF consists of:

- A set of **production rules** where the left-hand side is a non-terminal and the right-hand side is a string of terminals and/or non-terminals.
- The **start symbol**, which is the initial symbol of the grammar.

For example:

```
<expression> ::= <expression> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= ( <expression> ) | id
```

---

8. Pushdown Automata

A **Pushdown Automaton (PDA)** is a type of automaton that is used to recognize context-free languages. It is an extension of a finite automaton with a stack, which allows it to store an unbounded amount of information.

The PDA can be used to accept a language by emptying its stack when the entire input string is processed.

To construct a Pushdown Automaton (PDA) for the language $\mathbf{L = \{a^n b^n \mid n \geq 1\}}$, we need to design the PDA such that it accepts strings where the number of 'a's is equal to the number of 'b's, with the following conditions:

1. The string starts with one or more 'a's.
2. For every 'a' encountered, push an 'a' onto the stack.
3. For every 'b' encountered, pop one 'a' from the stack.
4. The string is accepted if, after processing all the input, the stack is empty and all symbols are consumed.

PDA Construction

We can represent the PDA as a 7-tuple $\mathbf{(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)}$:

- $\mathbf{Q}$: A finite set of states.
- $\mathbf{\Sigma}$: The input alphabet (the symbols that can appear in the input string).
- $\mathbf{\Gamma}$: The stack alphabet (the symbols that can be pushed onto or popped from the stack).
- $\mathbf{\delta}$: The transition function that describes the state transitions and stack operations.
- $\mathbf{q_0}$: The initial state of the PDA.
- $\mathbf{Z_0}$: The initial symbol on the stack.
- $\mathbf{F}$: The set of accept states.

Components of the PDA

1. $\mathbf{Q} = \{q_0, q_1, q_2\}$
   - q_0: Initial state, where we begin processing the input.
   - q_1: Intermediate state for processing 'a's.
   - q_2: Accept state, where the string is accepted after all input is consumed and the stack is empty.
2. $\mathbf{\Sigma} = \{a, b\}$
   - The input alphabet is composed of 'a' and 'b'.
3. $\mathbf{\Gamma} = \{a, Z_0\}$
   - The stack alphabet consists of 'a' (which we push and pop) and the initial stack symbol Z_0 (which marks the bottom of the stack).
4. $\mathbf{\delta}$: The transition function is defined as follows:
   - $\mathbf{\delta(q_0, a, Z_0) \rightarrow (q_0, aZ_0)}$: If we are in state q_0 and the input symbol is 'a' with the stack top symbol being Z_0, we push 'a' onto the stack and stay in state q_0.

- **$\delta(q_0, a, a) \rightarrow (q_0, aa)$**: If we are in state q_0 and the input symbol is 'a' with the stack top symbol being 'a', we push another 'a' onto the stack and stay in state q_0.
- **$\delta(q_0, b, a) \rightarrow (q_1, \varepsilon)$**: If we are in state q_0 and the input symbol is 'b' with the stack top symbol being 'a', we pop the 'a' from the stack and move to state q_1.
- **$\delta(q_1, b, a) \rightarrow (q_1, \varepsilon)$**: If we are in state q_1 and the input symbol is 'b' with the stack top symbol being 'a', we pop the 'a' from the stack and stay in state q_1.
- **$\delta(q_1, \varepsilon, Z_0) \rightarrow (q_2, Z_0)$**: If we are in state q_1 and the input is empty (i.e., we've consumed all the symbols) with the stack top symbol being Z_0, we move to the accept state q_2 and the stack remains unchanged.

5. **q_0**: The initial state.
6. **Z_0**: The initial stack symbol.
7. **F** = $\{q_2\}$: The accept state.

Key Difference between Finite Automata and Push Down Automata:

- **Finite Automaton (FA)**: Only has a finite set of states and cannot store any extra information beyond the current state.
- **Pushdown Automaton (PDA)**: Has a stack, allowing it to store an unbounded amount of information. This stack allows the PDA to handle languages with nested or recursive structures (like balanced parentheses or equal numbers of as and bs).

Example:

- **Finite Automaton Example**: Consider the language of strings containing at least one a (`L = {w | w contains at least one 'a'}`). A finite automaton can recognize this because it just needs to check if any 'a' appears in the input. The FA transitions between two states (one for seeing 'a' and one for not seeing 'a') and does not need any additional memory.
- **Pushdown Automaton Example**: Consider the language $L = \{a^n b^n \mid n \geq 1\}$ (equal numbers of as and bs). A finite automaton cannot recognize this because it cannot "count" the number of as and bs. However, a PDA can store an a on the stack every time it encounters one and pop it when it encounters a b. If the stack is empty at the end and the entire input string is processed, the PDA accepts the string.

---

9. Equivalence of Context-Free Languages and PDA

Context-free languages (CFLs) and **Pushdown Automata (PDA)** are equivalent in the sense that every context-free language can be recognized by a PDA.

A context-free grammar can be converted to a PDA, and vice versa. The PDA uses its stack to handle the non-context-sensitive features of the language (such as matching parentheses or balanced symbols), which is what makes it capable of recognizing CFLs.

---

10. Pumping Lemma for Context-Free Languages

The Pumping Lemma for Context-Free Languages is a property that helps prove whether a given language is not context-free. It states that for any context-free language, there exists a "pumping length" p such that any string w in the language of length at least p can be divided into five parts: `w = uvxyz`, where:

- `|vxy| ≤ p` (the length of vxy is at most p),
- `|vy| > 0` (the string vy is non-empty),
- `uv^n x y^n z` is in the language for all `n ≥ 0`.

This property is used to show that certain languages cannot be generated by context-free grammars.

---

11. Properties of Context-Free Languages

Context-free languages (CFLs) are a class of languages that can be generated by context-free grammars. They have certain unique properties that make them essential in formal language theory, especially in the context of compiler design and programming language syntax.

- **Closure Properties**: Context-free languages are closed under union, concatenation, and Kleene star, but not under intersection or difference.
- **Decidability**: Many problems, such as membership (whether a string belongs to a language) and emptiness (whether a language is empty), are decidable for context-free languages.
- **Parsing**: Parsing context-free languages is a central problem in compiler design. Techniques like **LL parsing** and **LR parsing** are used to parse context-free languages.

Important Differences Between Regular Languages and Context-Free Languages:

- **Regular Languages** (recognized by Finite Automata) cannot handle recursion or nested structures. They are simpler and only include patterns that can be described by finite state machines (e.g., `a*`, `ab*`).
- **Context-Free Languages** (recognized by Pushdown Automata) can handle complex patterns like matching parentheses or recursive structures, requiring the use of a stack.
- **FA recognizes Regular Languages**, which are simpler and do not involve recursion or nested patterns.
- **PDA recognizes Context-Free Languages (CFLs)**, which are more complex and include patterns like balanced parentheses, matched numbers of `a`s and `b`s, palindromes, and nested expressions.

Conclusion

- **Context-Free Grammar (CFG)** is used to define context-free languages, where each production rule has a single non-terminal on the left-hand side.
- **Derivation trees** and **parse trees** are used to represent the structure of strings generated by a CFG.
- A grammar is **ambiguous** if there is more than one parse tree for a string.
- **Normal forms** like CNF, GNF, and BNF provide standardized ways of expressing grammars.
- **Pushdown Automata (PDA)** recognize context-free languages, and **PDAs** are equivalent to **context-free grammars**.
- The **Pumping Lemma** is used to prove that certain languages are not context-free.

6.3 Turing Machines (TM)

In this section, we will explore **Turing Machines (TM)**, which are fundamental models of computation. A Turing Machine is an abstract machine that can simulate any algorithm's logic and is the basis for the theory of computation. It serves as a powerful model for analyzing the limits of what can be computed.

---

1. Introduction to Turing Machines

A **Turing Machine (TM)** is a theoretical computing machine invented by Alan Turing in 1936. It is a mathematical model that defines a hypothetical machine capable of performing any computation that can be algorithmically described. The machine operates on an infinite tape and follows predefined rules for moving along the tape and changing symbols based on the state it is in.

A Turing Machine consists of:

- **An infinite tape** divided into cells, each of which holds a symbol.
- **A tape head** that can read and write symbols on the tape and move left or right.
- **A finite set of states**, including a special start state and one or more accepting or halting states.
- **A transition function** that dictates the machine's behavior, defining how it moves between states and modifies the tape.

---

2. Notations of Turing Machine

The Turing Machine is formally defined as a 7-tuple:

$M = (Q, \Sigma, \Gamma, \delta, q_0, q\_accept, q\_reject)$

Where:

- Q is a finite set of states.
- Σ is the input alphabet (symbols the machine can read).
- Γ is the tape alphabet (includes symbols from Σ and the blank symbol).
- δ is the transition function:
  $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, defining state transitions, tape modifications, and head movement (Left or Right).
- q_0 is the initial state.
- q_accept is the accepting state.
- q_reject is the rejecting state.

---

3. Acceptance of a String by a Turing Machine

A Turing Machine accepts a string if, starting from the initial state and reading the string from the input tape, the machine eventually reaches the **accept state** after following the rules in the transition function. If the machine reaches the **reject state** or halts without accepting, the string is rejected.

The process of acceptance involves:

1. Reading the symbols on the tape.
2. Transitioning between states and modifying the tape content.
3. Reaching the accepting state, which signifies the string is accepted.

---

4. Turing Machine as a Language Recognizer

A Turing Machine is considered a **language recognizer** because it can decide whether a given string belongs to a particular language. A language LL is said to be **Turing-**

recognizable (or **recursively enumerable**) if there exists a Turing Machine that accepts every string in LL and either rejects or runs forever on strings not in LL.

---

5. Turing Machine as a Computing Function

A Turing Machine can also be viewed as a **computing function**. In this context, a Turing Machine computes a function by transforming an input string into an output string. For example, a Turing Machine can compute the successor of a number or perform mathematical operations by modifying the tape and transitioning between states according to its transition function.

---

6. Turing Machine as an Enumerator of Strings of a Language

A Turing Machine can be used as an **enumerator** for a language, meaning it can generate all strings in the language, one by one. The Turing Machine enumerates the strings by printing them on the tape and then moving to the next string after halting. A language is **recursively enumerable** if a Turing Machine can enumerate all its strings.

---

7. Turing Machine with Multiple Tracks

A **multi-track Turing Machine** is an extension of the basic Turing Machine where the tape is divided into several tracks, each of which can hold a symbol. The machine can move the tape head independently on each track, which allows it to handle multiple pieces of information simultaneously. This extension helps model more complex computations.

---

8. Turing Machine with Multiple Tapes

A **multi-tape Turing Machine** has several tapes and heads, each operating independently. This extension allows more efficient computation compared to a single-tape machine, but it doesn't change the computational power of the Turing Machine. A multi-tape machine can simulate a single-tape machine and vice versa, meaning they are **equivalent** in terms of computational power.

---

9. Non-Deterministic Turing Machines (NDTM)

A **Non-Deterministic Turing Machine (NDTM)** is a variation of the Turing Machine where the transition function is not deterministic. At each step, the NDTM can transition to multiple possible next states for a given input symbol, and it can **choose** one of these transitions. An NDTM is used to recognize languages that are **non-deterministically** decidable.

- **Acceptance**: An NDTM accepts a string if there exists at least one sequence of transitions leading to an accepting state.

NDTMs are important because they are used to define the class of **NP** problems in computational complexity theory.

---

10. Church-Turing Thesis

The **Church-Turing Thesis** posits that anything that can be computed algorithmically can be computed by a Turing Machine. This thesis implies that Turing Machines are as powerful as any computational device, and they can simulate any computation that can be performed by modern computers. The thesis serves as a foundation for the field of **computational theory**.

---

11. Universal Turing Machine

A **Universal Turing Machine (UTM)** is a Turing Machine that can simulate any other Turing Machine. It takes as input the description of another Turing Machine (encoded as a string) and its input tape, and it simulates the computation of that Turing Machine on the given input. The existence of the UTM shows that a single machine can perform any computation, given the right instructions.

---

12. Computational Complexity

Computational complexity refers to the study of the resources required for a Turing Machine to solve a problem. The two primary resources are **time** and **space**.

- **Time complexity**: The number of steps (or transitions) a Turing Machine takes to process an input.
- **Space complexity**: The amount of tape (memory) used by a Turing Machine during computation.

Time and space complexities are used to classify problems into different complexity classes, such as **P**, **NP**, and **NP-complete**.

---

13. Intractability

A problem is said to be **intractable** if it is **computationally expensive** to solve, often because it has high time complexity (e.g., exponential time). Intractable problems cannot be solved in a reasonable amount of time for large inputs. Some examples of intractable problems include the **Traveling Salesman Problem** and **Integer Factorization**.

---

14. Reducibility

Reducibility is the concept of transforming one problem into another. If we can reduce problem AA to problem BB, it means that solving BB would also solve AA. This is important in computational complexity theory, as it helps classify problems based on their relative difficulty.

- **Polynomial-time reducibility**: If problem AA can be reduced to problem BB in polynomial time, we say AA is **polynomial-time reducible** to BB.
- **NP-completeness**: A problem is **NP-complete** if it is both in NP and as hard as any other problem in NP, meaning every NP problem can be reduced to it in polynomial time.

Conclusion

- **Turing Machines (TM)** are abstract mathematical models of computation that define the limits of what can be computed algorithmically.
- **Non-Deterministic Turing Machines (NDTM)** and **Universal Turing Machines (UTM)** extend the model to recognize non-deterministic languages and simulate other TMs.
- **Church-Turing Thesis** suggests that anything computable by an algorithm can be computed by a TM.
- **Computational complexity** studies the time and space required to solve problems, with the goal of understanding intractable problems and their reducibility.

6.4 Introduction to Computer Graphics

Computer graphics is the field of computer science that focuses on generating, manipulating, and displaying visual images using computers. It combines elements of mathematics, physics, and art to create visual representations of data, virtual environments, and multimedia content. This section covers the basics of computer graphics, graphics hardware, software, and input/output devices.

---

1. Overview of Computer Graphics

Computer graphics can be divided into two primary categories:

- **2D Graphics**: Representations of objects in two dimensions, such as images and drawings.
- **3D Graphics**: Representations of objects in three-dimensional space, often used in gaming, simulations, and design.

Graphics are essential in various applications, including:

- **Entertainment**: Movies, video games, and simulations.
- **Engineering**: CAD (Computer-Aided Design) and 3D modeling.
- **Scientific Visualization**: Displaying complex data and simulations.
- **Web Design and User Interfaces**: Visual design of websites and apps.

The process of creating graphics involves both **rendering** (creating the image from models) and **displaying** (outputting the image on a screen or print).

---

2. Graphics Hardware

The hardware used in computer graphics plays a crucial role in generating and displaying images efficiently. It includes various display technologies, devices, and processors.

1. **Display Technology**

Displays are devices used to visually present information generated by a computer. There are two main types of display technologies used in computer graphics:

- **Raster-Scan Displays**: These displays use a grid of pixels to represent the image. The screen is refreshed by scanning the grid line-by-line (raster scan). Common examples include monitors and televisions.
  - Each pixel has its color and intensity, and the entire screen is refreshed rapidly to create an image.

- **Vector Displays**: Unlike raster displays, vector displays create images by directly drawing lines between points on the screen. They are less common today but were widely used in early computer graphics for drawing simple shapes.
  - Vector displays have higher clarity for lines but are limited to drawing lines and geometric shapes.

2. **Architecture of Raster-Scan Displays**

Raster-scan displays use a **cathode ray tube (CRT)** or **LCD screen** to display images by scanning the entire screen in a raster pattern. The basic process includes:

- The electron gun or liquid crystals scan across the screen in a series of horizontal lines (scan lines).
- Each line is divided into **pixels** (picture elements), which are the smallest addressable units of the screen.
- The image is drawn by changing the color and intensity of individual pixels as the scan progresses.

3. **Display Processors**

Display processors are dedicated hardware components responsible for controlling the rendering of images to the screen. These processors work with the **framebuffer**, which is a memory area that holds the image data to be displayed. They can perform operations like:

- Pixel manipulation (color, brightness, etc.)
- Handling resolution changes
- Interfacing with the graphics card

Graphics processors (GPUs) are specialized for high-speed rendering and computation tasks involved in 3D graphics, image processing, and video acceleration.

4. **Output Devices**

- **Monitors/Displays**: These devices show the visual output of computer systems. They can be CRT, LCD, LED, OLED, etc.
- **Printers**: Though primarily for static images, printers can produce graphics on paper, often used in technical design and documentation.
- **Projectors**: Used for large-scale displays, such as presentations, virtual reality, or collaborative workspaces.

5. **Input Devices**

Input devices allow users to interact with computer graphics systems, such as drawing, selecting, or manipulating graphical objects:

- **Mouse**: Used for pointing and selecting objects.
- **Keyboard**: Allows user input for commands and text.
- **Graphics Tablet**: A specialized device for precise drawing and design.
- **Trackball**: A pointing device used for manipulating graphics.
- **Touchscreen**: Allows direct interaction with graphical elements through touch.

---

3. Graphics Software and Software Standards

Graphics software is essential for creating, manipulating, and displaying images and models. It can be categorized into:

- **Rendering Software**: Tools that generate images from 3D models (e.g., Blender, Maya).
- **CAD Software**: Used for technical drawings, architecture, and engineering (e.g., AutoCAD).

- **Image Editing Software**: Tools like Photoshop and GIMP for editing raster images.
- **Vector Graphics Software**: Tools like Adobe Illustrator for creating and editing vector-based images.

1. **Software Standards**

Software standards ensure compatibility and consistency across graphics systems. Key standards include:

- **OpenGL**: A widely-used, cross-platform API for rendering 2D and 3D graphics. It provides functions for drawing shapes, handling textures, and applying lighting effects.
- **DirectX**: A collection of APIs developed by Microsoft for multimedia applications, including graphics rendering.
- **Vulkan**: A low-level, high-performance API for graphics rendering, developed by the Khronos Group (the same group behind OpenGL).
- **SVG (Scalable Vector Graphics)**: An XML-based vector graphics format used for web design and other applications that require resolution-independent images.
- **PDF (Portable Document Format)**: Supports vector-based images, making it useful for displaying graphical designs and technical drawings.
- **WebGL**: A web-based API for rendering interactive 3D graphics within a browser without the need for plugins.

Conclusion

- **Computer graphics** involves generating and displaying images, and it is used in many fields like entertainment, science, and engineering.
- **Graphics hardware** includes components like raster-scan displays, vector displays, display processors, and various input/output devices.
- **Graphics software** includes rendering, CAD, and image editing software, and there are various software standards like OpenGL and DirectX that define how graphics are created and displayed on computers.

6.5 Two-Dimensional Transformation

Two-dimensional (2D) transformations are fundamental operations used to manipulate and modify graphical objects in a 2D coordinate system. These transformations are applied to shapes, images, or objects to change their position, orientation, size, or other properties. They form the basis for rendering graphics in computer systems and are commonly used in applications such as gaming, CAD, and computer graphics.

---

1. Two-Dimensional Translation

Translation is the process of moving an object from one location to another within the 2D coordinate plane without changing its shape, size, or orientation. The object is shifted by a specified distance along the X and Y axes.

- **Mathematical Representation**:
  - If
    $$(x, y)$$
    is the original point, and
    $$(t_x, t_y)$$

is the translation vector, then the new point

$$(x', y')$$

after translation is given by:
  * $x' = x + t_x$
  * $y' = y + t_y$

- **Transformation Matrix**:

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

---

2. Two-Dimensional Rotation

Rotation involves turning an object around a fixed point (typically the origin of the coordinate system) by a specified angle. Positive angles correspond to counterclockwise rotation, and negative angles correspond to clockwise rotation.

- **Mathematical Representation**: If the point

$$(x, y)$$

is rotated by an angle (

$$\theta$$

), the new point

$$(x', y')$$

is given by:
  - $x' = x\cos\theta - y\sin\theta$
  - $y' = x\sin\theta + y\cos\theta$

- **Transformation Matrix**:

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

---

3. Two-Dimensional Scaling

Scaling alters the size of an object by a scale factor along the X and Y axes. Scaling can be uniform (the same factor for both axes) or non-uniform (different factors for X and Y axes).

- **Mathematical Representation**: If

$$(x, y)$$

is the original point and

$$(s_x)$$

and

$$(s_y)$$

are the scaling factors along the X and Y axes, then the new point

$$(x', y')$$

is given by:

- $x' = x \cdot s_x$
- $y' = y \cdot s_y$

- **Transformation Matrix**:

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

---

4. Two-Dimensional Reflection

Reflection involves flipping an object over a line (axis) such as the X-axis, Y-axis, or a diagonal line, producing a mirror image.

- **Reflection over the X-axis**:
  - **Mathmatical Representations**
    - $x' = x$
    - $y' = -y$
  - **Transformation matrix:**

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Reflection over the Y-axis**:
  - **Mathmatical Representations**
    - $x' = -x$
    - $y' = y$
    - **Transformation matrix:**

$$R_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Reflection over the line (y = x)**:
  - **Mathmatical Representations**
    - $x' = y$
    - $y' = x$
  - **Transformation matrix:**

$$R_{xy} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

---

5. Two-Dimensional Shear Transformation

Shear transformation distorts an object by shifting its points in one direction (horizontal or vertical) based on its position. It is often used in graphics for creating effects like slanting or skewing.

- **Mathematical Representation**:
  - **Horizontal shear** (shifting X coordinates):
    - $x' = x + sh_x \cdot y$
    - $y' = y$
  - **Vertical shear** (shifting Y coordinates):

    * $x' = x$
    * $y' = y + sh_y \cdot x$
- **Transformation Matrix**:
  - **Horizontal shear**:

$$H_{sh_x} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  - **Vertical shear**:

$$H_{sh_y} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

---

6. Two-Dimensional Composite Transformation

A **composite transformation** involves applying multiple transformations to an object sequentially. This can include any combination of translation, rotation, scaling, and reflection.

The composite transformation matrix is the product of individual transformation matrices. For example, applying **scaling**, then **rotation**, and finally **translation** can be represented by the following:

$$T_{composite} = T \cdot R \cdot S$$

This allows complex transformations to be represented as a single operation, simplifying computation and rendering.

---

7. 2D Viewing Pipeline

The **2D viewing pipeline** is a process that converts a 3D scene or world coordinates into 2D screen coordinates, which can then be displayed on the screen. The pipeline typically involves:

1. **World-to-View Transformation**: Converts the world coordinate system to the camera or view coordinate system.
2. **Projection Transformation**: Projects the 3D scene onto a 2D plane.
3. **Clipping**: Removes objects or parts of objects that fall outside the viewing area.
4. **Window-to-Viewport Transformation**: Maps the 2D coordinates to screen coordinates.

---

8. Clipping

Clipping is the process of removing parts of an object that lie outside a specified region, often referred to as the clipping window. This is critical in computer graphics to ensure that only the visible portion of objects is rendered.

1. **Cohen-Sutherland Line Clipping Algorithm**

The **Cohen-Sutherland algorithm** is used for line clipping. It divides the space into regions and assigns each endpoint a 4-bit code (outcode). Depending on the outcodes of the endpoints, the line can be:

- Completely inside the clipping window.
- Completely outside the clipping window.
- Partially inside the clipping window (requiring further clipping).

2. **Liang-Barsky Line Clipping Algorithm**

The **Liang-Barsky algorithm** is another line clipping algorithm that is more efficient than Cohen-Sutherland for axis-aligned rectangles. It works by testing the parameterized equation of the line against the clipping boundaries.

Conclusion

In 2D graphics, transformations are used to manipulate objects by changing their position, orientation, and size. The core transformations include translation, rotation, scaling, reflection, and shear. These transformations can be combined in a composite transformation to produce more complex effects. The **2D viewing pipeline** allows 3D scenes to be converted into 2D representations for display on a screen. **Clipping** algorithms, such as Cohen-Sutherland and Liang-Barsky, are used to remove parts of objects that fall outside the defined view area.

6.6 Three-Dimensional Transformation

Three-dimensional (3D) transformations are used to manipulate 3D objects in a 3D coordinate system. These transformations can change the position, orientation, size, or shape of objects in 3D space. Like 2D transformations, 3D transformations are critical in computer graphics for applications like 3D modeling, animation, and gaming.

---

1. Three-Dimensional Translation

Translation in 3D is the process of moving an object from one location to another along the X, Y, and Z axes. The object is shifted by a specified distance in each direction without altering its shape or orientation.

- **Mathematical Representation**: If
  $$(x, y, z)$$
  is the original point, and
  $$(tx, ty, tz)$$
  is the translation vector, the new point
  $$(x', y', z')$$
  after translation is:
  - $x' = x + tx$
  - $y' = y + ty$
  - $z' = z + tz$
- **Transformation Matrix**:

$$T = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

2. Three-Dimensional Rotation

Rotation in 3D involves rotating an object around one of the three axes (X, Y, or Z). A point in 3D space is rotated by a specified angle around the axis of rotation.

- **Rotation about the X-axis**:
  - $x' = x$
  - $y' = y \cdot \cos\theta - z \cdot \sin\theta$
  - $z' = y \cdot \sin\theta + z \cdot \cos\theta$
- **Rotation about the Y-axis**:
  - $x' = x \cdot \cos\theta + z \cdot \sin\theta$
  - $y' = y$
  - $z' = -x \cdot \sin\theta + z \cdot \cos\theta$
- **Rotation about the Z-axis**:
  - $x' = x \cdot \cos\theta - y \cdot \sin\theta$
  - $y' = x \cdot \sin\theta + y \cdot \cos\theta$
  - $z' = z$
- **Rotation Matrices**:
  - Rotation about the X-axis:

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

  - Rotation about the Y-axis:

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

  - Rotation about the Z-axis:

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

---

3. Three-Dimensional Scaling

Scaling in 3D modifies the size of an object along the X, Y, and Z axes. It can be uniform (the same factor for all axes) or non-uniform (different scaling factors for each axis).

- **Mathematical Representation**: If

$$(x, y, z)$$

  is the original point and

$$(sx, sy, sz)$$

  are the scaling factors along the X, Y, and Z axes, then the new point

$$(x', y', z')$$

  is:

  - $x' = x \cdot sx$
  - $y' = y \cdot sy$

- $z' = z \cdot sz$

- **Transformation Matrix**:

$$S = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

4. Three-Dimensional Reflection

Reflection in 3D mirrors an object across a plane. The reflection can be done across the XY, YZ, or XZ plane, or across an arbitrary plane.

- **Reflection across the XY plane**:
  - **Mathematical Expression:**
    - $x' = x$
    - $y' = y$
    - $z' = -z$
  - **Transformation matrix**:

$$R_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Reflection across the YZ plane**:
  - **Mathematical Expression:**
    - $x' = -x$
    - $y' = y$
    - $z' = z$
  - **Transformation matrix**:

$$R_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

5. Three-Dimensional Shear Transformation

Shear transformation in 3D distorts an object by shifting points in one direction based on their position in another direction. This transformation can be applied to all three axes.

- **Mathematical Representation**:
  - **Shear along the X-axis**:
    - $x' = x + sh_x \cdot y + sh_x z \cdot z$
    - $y' = y$
    - $z' = z$
  - **Shear along the Y-axis**:
    - $x' = x$
    - $y' = y + sh_y \cdot x + sh_y z \cdot z$
    - $z' = z$
  - **Shear along the Z-axis**:
    - $x' = x$

    * $y' = y$
    * $z' = z + sh_z \cdot x + sh_z y \cdot y$
- **Transformation Matrix**:
  - **Shear along the X-axis**:

$$H_{sh_x} = \begin{bmatrix} 1 & sh_x & sh_x z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - **Shear along the Y-axis**:

$$H_{sh_y} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ sh_y & 1 & sh_y z & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - **Shear along the Z-axis**:

$$H_{sh_z} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ sh_z & sh_z y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

6. Three-Dimensional Composite Transformation

Composite transformation in 3D is the combination of multiple transformations (translation, rotation, scaling, reflection, shear) applied sequentially to an object. The resulting transformation matrix is obtained by multiplying the individual transformation matrices together.

For example, applying **scaling**, then **rotation**, then **translation**:

$$T_{composite} = T \cdot R \cdot S$$

---

7. 3D Viewing Pipeline

The **3D viewing pipeline** transforms a 3D world into a 2D screen space, similar to the 2D pipeline, but with additional steps to account for the third dimension.

1. **Modeling Transformation**: Transforms the object from model coordinates to world coordinates.
2. **View Transformation**: Converts the world coordinates to camera (view) coordinates.
3. **Projection Transformation**: Projects the 3D world onto a 2D plane (screen space).
4. **Clipping**: Removes objects outside the view frustum.
5. **Viewport Transformation**: Converts the 2D coordinates to screen coordinates.

---

8. Projection Concepts

- **Orthographic Projection**:
  - Objects are projected onto a 2D plane along parallel lines. There is no perspective distortion; objects remain the same size regardless of distance from the camera.

  - Used for architectural drawings, engineering, etc.
- **Parallel Projection**:
  - Similar to orthographic but can also involve objects being projected along non-orthogonal directions.
- **Perspective Projection**:
  - Mimics the human eye's perception, where objects appear smaller as they move farther from the camera, creating a sense of depth.
  - Defined by a **vanishing point** and **viewing distance**.

Conclusion

Three-dimensional transformations are the cornerstone of 3D graphics, enabling the manipulation of objects in a 3D coordinate system for applications such as modeling, animation, and visualization.

- **Translation, Rotation, Scaling, Reflection, and Shear** allow for fundamental changes in an object's position, orientation, size, and shape.
- **Composite transformations** efficiently combine multiple operations into a single matrix, streamlining complex workflows.
- **The 3D Viewing Pipeline** ensures that the transformation from 3D space to a 2D screen accurately represents the intended scene.
- **Projection concepts** like orthographic and perspective projection play a vital role in rendering realistic or technical views.

Mastering these transformations is essential for creating immersive and visually compelling 3D environments, driving innovation in fields like gaming, simulation, and virtual reality.