CHAPTER 1.

Introduction

1.1 Computer Graphics using OPENGL:

Open Graphics Library (OpenGL) is a cross-language (language independent), cross-platform (platform-independent) API for rendering 2D and 3D Vector Graphics(use of polygons to represent image). OpenGL API is designed mostly in hardware.

Cg is a high-level shading language developed by NVIDIA in collaboration with Microsoft for programming vertex and pixel shaders. It stands for "C for Graphics" because it is derived from the syntax and structure of the C programming language, making it relatively easy for C/C++ programmers to learn and use.

Key Features:

- High-Level Abstraction: Allows developers to write shaders (programs used in graphics processing units) in a C-like language rather than directly in assembly language.
- Cross-Platform Support: Originally developed for use with NVIDIA GPUs, Cg is supported across multiple platforms and can compile shaders for different GPU architectures.
- Integration: Cg integrates well with other graphics APIs like OpenGL and Direct3D, making it versatile for different programming environments.

1.2OpenGL (Open Graphics Library)

OpenGL is a cross-platform graphics API that provides a set of functions for rendering 2D and 3D graphics. It is widely used in applications ranging from video games to scientific visualization and virtual reality.

Key Features:

- Cross-Platform: OpenGL is supported on multiple platforms including Windows, macOS, Linux, and mobile operating systems.
- Hardware Acceleration: Provides access to hardware-accelerated graphics rendering, leveraging the capabilities of GPUs for faster and more efficient rendering.
- Extensible: OpenGL is designed to be extended through extensions and vendor-specific features, allowing developers to access advanced rendering capabilities.
- State Machine: Operates on a state machine paradigm where the current state dictates how rendering commands affect the graphics output.

1.3 Cg and OpenGL Integration:

- Cg shaders can be used within an OpenGL application to define how vertices and pixels are processed during rendering.
- Cg shaders are compiled into a format that OpenGL understands, allowing developers to harness the power of GPU acceleration while using a high-level shading language.

In summary, Cg and OpenGL are complementary technologies used in graphics programming, with Cg providing a high-level language for writing shaders and OpenGL providing the framework for rendering those shaders on different platforms. Their integration allows developers to create visually compelling applications and simulations with efficient graphics processing capabilities.

Cg and OpenGL work together to provide a powerful framework for graphics programming:

- **Shader Compilation:** Cg shaders are written using the Cg language and then compiled into a format that OpenGL can understand. This allows developers to define custom vertex and fragment shaders to control how geometry and pixels are rendered.
- **Rendering Pipeline:** OpenGL provides a flexible rendering pipeline where developers can use Cg shaders to customize various stages of rendering:
  - **Vertex Processing:** Cg vertex shaders can manipulate vertex positions, apply transformations, and compute per-vertex attributes.
  - **Fragment Processing:** Cg fragment shaders can calculate colors, apply textures, perform lighting calculations, and more.
- **Efficiency and Performance:** By utilizing Cg shaders within OpenGL, developers can leverage GPU hardware acceleration for efficient rendering. This combination enables complex graphical effects and high-performance graphics applications.
- **Cross-Platform Compatibility:** Since both Cg and OpenGL are designed to be cross-platform, applications developed using these technologies can run on different operating systems and hardware configurations without significant modifications.

1.4 Practical Use Cases

- **Video Games:** Cg and OpenGL are extensively used in game development for rendering 3D graphics, applying visual effects (such as lighting and shadows), and optimizing performance for real-time gameplay.
- **Simulation and Visualization:** Applications in fields like scientific visualization, virtual reality (VR), and computer-aided design (CAD) benefit from Cg and OpenGL's ability to render complex models and simulate realistic environments.
- **Education and Research:** These technologies are also used in educational settings and research projects to study graphics algorithms, explore visual computing concepts, and develop interactive simulations.

1.5 Image Processing using OPENCV:

OpenCV (Open Source Computer Vision Library) is a powerful open-source library primarily aimed at real-time computer vision tasks. It was originally developed by Intel and later supported by Willow Garage and now by Itseez. It's written in C++ and has bindings for Python, making it accessible and widely used across different programming environments.

Key Features of OpenCV:

- **Cross-Platform:** OpenCV supports various operating systems including Windows, macOS, Linux, and mobile platforms like Android and iOS. This cross-platform capability makes it versatile for a wide range of applications.
- **Extensive Functionality:** OpenCV offers a comprehensive suite of functions and algorithms for image processing and computer vision tasks. These include:
- **Image I/O:** Loading, saving, and displaying images in various formats.

Basic Image Operations: Manipulating images at the pixel level, such as resizing, cropping, rotating, and flipping.

- **Image Filtering and Enhancement:** Applying filters like Gaussian blur, sharpening, edge detection, and histogram equalization.
- **Feature Detection:** Finding key points and descriptors in images using algorithms like Harris corner detection, SIFT (Scale-Invariant Feature Transform), SURF (Speeded-Up Robust Features), and ORB (Oriented FAST and Rotated BRIEF).
- **Object Detection and Tracking:** Methods like Haar cascades, HOG (Histogram of Oriented Gradients), and deep learning-based techniques (using frameworks like TensorFlow and PyTorch).
- **Camera Calibration and 3D Reconstruction:** Tools for calibrating cameras, rectifying stereo images, and reconstructing 3D scenes from multiple viewpoints.
- **Machine Learning Integration:** OpenCV integrates with machine learning libraries for tasks such as image classification, object segmentation, and facial recognition.
- **Performance Optimization:** OpenCV leverages hardware acceleration capabilities (e.g., SIMD instructions, CUDA for NVIDIA GPUs) for faster computation, crucial for real-time applications.
- **Community and Documentation:** Being open-source, OpenCV benefits from a large community of developers contributing to its development and maintenance. It has extensive documentation, tutorials, and examples to facilitate learning and usage.

CHAPTER 2.
Creating a Dynamic Cityscape and celestial bodies Simulation

This project aims to create a dynamic graphical representation of a cityscape using PyOpenGL and GLUT in Python. The cityscape changes based on the time of day, providing a visually appealing and realistic simulation of a day-night cycle. The project combines various graphical elements, including buildings, trees, celestial objects (sun and moon), birds, clouds, stars, and a timestamp, to create a rich and engaging visual experience.

2.1 Key Components and Functions

PyOpenGL and GLUT

- **PyOpenGL**: This is the Python binding for OpenGL, a powerful graphics library that allows for the creation of complex 2D and 3D graphics. OpenGL provides a wide range of functions for rendering graphics, handling colors, and managing graphical transformations.
- **GLUT (OpenGL Utility Toolkit)**: GLUT is a utility toolkit that simplifies the implementation of programs using OpenGL by providing functions for window management, user input, and other tasks related to the graphical user interface.

NumPy

- **NumPy**: This is a fundamental library for numerical operations in Python. It is used in this project for handling arrays and performing numerical computations, such as generating random positions for birds and interpolating colors.

Time

- **Time**: This standard Python module is used for time-related operations, such as keeping track of the current time and updating the scene based on the progression of time.

2.2 Project Overview

The project simulates a cityscape that evolves throughout the day, transitioning smoothly from morning to afternoon, evening, and night. This is achieved by updating the background color, changing the positions of the sun and moon, and modifying the visibility of stars and other elements based on the time of day.

Window and Time Initialization

- **Window Dimensions**: The graphical window is set to a size of 800x600 pixels.
- **Initial Time**: The simulation starts at 12.0 hours (noon), providing a baseline for the progression of time.

Bird Positions

- **Birds**: The cityscape includes five birds whose positions are randomly initialized within the top half of the screen. These birds add a dynamic and lively element to the scene.

Drawing and Animation

The project employs a variety of drawing functions to render different elements of the cityscape:

- **Points and Lines**: Functions such as draw_points and MidPointLine are used to draw individual points and lines, which form the basic building blocks for more complex shapes.
- **Zone Conversion**: To handle line drawing in different octants efficiently, the project includes functions like findZone, ZoneZeroConversion, and zero_to_original_zone.
- **Color Interpolation**: The interpolate_color function smoothly transitions between colors to simulate changes in lighting throughout the day.
- **Background Color**: The BackGroundColour function sets the background color based on the current time, providing a visual cue for the time of day.

Rendering Buildings, Trees, and Other Elements

- **Buildings**: The draw_building function creates buildings with specified dimensions and colors, including details like windows and doors.
- **Trees**: The draw_tree function adds trees to the scene, enhancing the visual complexity and natural feel of the cityscape.
- **Birds**: The draw_bird function draws birds at their respective positions, with an optional size scaling factor.
- **Circles**: The draw_circle function renders filled circles, which are used for drawing elements like the sun and moon.
- **Celestial Objects**: The draw_sun_and_moon function updates the position and appearance of the sun or moon based on the time of day.
- **Clouds and Stars**: The draw_cloud and draw_stars functions add clouds and stars to the scene, further enhancing the realism of the cityscape.
- **Timestamp**: The draw_timestamp function displays the current time on the screen, providing a real-time indication of the simulated time.

Time Management and Animation

- **Updating Time**: The update_time function increments the current time, simulating the passage of time in the cityscape.
- **Day-Night Cycle**: The day_night function determines whether it is day or night based on the current time, affecting the visibility of stars and other elements.

2.3 OpenGL Setup

- **Initialization**: The init function sets up the OpenGL environment, including the background color and orthographic projection.
- **Display Update**: The update function schedules regular updates to the display, ensuring smooth animation and timely updates of the scene.

Main Function

The main function initializes the GLUT framework, sets up the display mode, window size, and position, and starts the main loop. This function brings together all the components and functions, creating a cohesive and dynamic graphical application.

In summary, this project showcases the power of PyOpenGL and GLUT for creating dynamic and interactive graphical applications in Python. By combining various elements and smoothly transitioning between different times of the day, it provides a compelling visual representation of a cityscape that evolves over time.

CHAPTER 3.

System Requirements and Specifications

This chapter details the hardware and software requirements necessary to implement and run the cityscape simulation project. Ensuring that the system meets these requirements will help achieve optimal performance and a seamless user experience.

3.1 Hardware Requirements

1. **Processor**:
   - A multi-core processor with at least 2.0 GHz speed is recommended to handle the computational tasks efficiently.
2. **Memory**:
   - A minimum of 4 GB RAM is required. However, 8 GB or more is recommended for better performance, especially when dealing with graphical rendering.
3. **Graphics Card**:
   - A dedicated graphics card that supports OpenGL 2.0 or higher is recommended to render graphics smoothly.
4. **Storage**:
   - At least 100 MB of free disk space for installing Python, required libraries, and storing project files.
5. **Display**:
   - A monitor with a minimum resolution of 1280x720 pixels. A higher resolution monitor is recommended for better visual experience.
6. **Input Devices**:
   - A standard keyboard and mouse for interacting with the simulation.

3.2 Software Requirements

1. **Operating System**:
   - Windows 7 or later, macOS 10.10 or later, or a recent Linux distribution. The software should support Python and OpenGL.

   - Python 3.6 or later is required to run the code. Python can be downloaded from the official Python website.

Python:

- Python 3.6 or later is required to run the code. Python can be downloaded from the official Python website.

1. **Python Libraries**:
   - **PyOpenGL**: Required for OpenGL bindings in Python.
     - Installation: pip install PyOpenGL PyOpenGL_accelerate
   - **GLUT**: Required for handling windowing and user input.
     - Installation: Typically comes with PyOpenGL. Alternatively, you can install it via your system's package manager.
   - **NumPy**: Required for numerical operations and managing bird positions.
     - Installation: pip install numpy
2. **Development Environment**:
   - A code editor or IDE such as Visual Studio Code, PyCharm, or any other text editor

3.3 Installation Instructions

1. **Python Installation**:
   - Download and install Python from the official Python website. Ensure that you add Python to your system's PATH during installation.
2. **Library Installation**:
   - Open a command prompt or terminal.
   - Install the required libraries using pip:

     pip install PyOpenGL PyOpenGL_accelerate numpy

3.GLUT Installation:
   - For Windows:
     - GLUT can be installed via the freeglut library. Download it from freeglut official website and follow the installation instructions.
   - For macOS:
     - GLUT is available through XQuartz. Download and install XQuartz from XQuartz official website.
   - For Linux:
     - Install GLUT using your package manager. For example, on Ubuntu:

       sudo apt-get install freeglut3-dev

4. **Project Files**:
   - Download or clone the project repository from the provided source. Ensure all the necessary files are in the project directory

3.4 Running the Project

1. **Open Command Prompt or Terminal**:
   - Navigate to the directory where the project files are stored.
2. **Run the Python Script**:
   - Execute the main script using Python:

     python cityscape.py

     This will start the simulation, and a window will open displaying the cityscape.
3. **Interacting with the Simulation**:
   - Use the mouse to interact with the simulation. Left-clicking will trigger bird movements.

By ensuring the system meets the above requirements and following the installation instructions, the cityscape simulation project should run smoothly and efficiently.

CHAPTER 4.

Methodology

The project aims to create a dynamic cityscape simulation that smoothly transitions between day and night, visualizing changes in the environment over a 24-hour cycle. This is achieved using OpenGL for rendering 2D graphics and GLUT for handling window management and user input. The simulation involves various visual elements including buildings, trees, the sun, the moon, clouds, stars, and birds, each contributing to the realism and aesthetic of the scene. The transitions are controlled by a time variable (current_time) which ranges from 0 to 24, representing the hours of the day. As the current_time changes, the colors of the sky, the visibility of the sun and moon, the presence of stars and clouds, and other environmental features are dynamically adjusted to reflect the corresponding time of day. This methodology outlines the design, implementation, and integration of these components to create a seamless and engaging visual experience.

4.1 Initialization Functions

- **initialize_bird_positions()**: Initializes the positions of birds randomly within the top half of the screen.

Drawing Functions

- **draw_points(x, y)**: Draws a single point at coordinates (x, y).
- **findZone(x0, y0, x1, y1)**: Determines the zone in which a line segment lies based on its endpoints.
- **ZoneZeroConversion(zone, x, y)** and **zero_to_original_zone(zone, x, y)**: Convert coordinates between different zones and their original orientation.
- **MidPointLine(zone, x0, y0, x1, y1)**: Draws a line between two points using the midpoint line algorithm.
- **interpolate_color(color1, color2, factor)**: Interpolates between two colors based on a given factor.
- **BackGroundColour(x)**: Determines the background color based on the time of day.
- **draw_building(x0, y0, width, height, color)**: Draws a building with specified dimensions and color.
- **draw_tree(x0, y0)**: Draws a stylized tree.
- **draw_bird(x, y, size=1.1)**: Draws a bird with an optional size scaling factor.
- **draw_circle(cx, cy, radius)**: Draws a filled circle.
- **draw_sun_and_moon(time)**: Draws the sun or the moon based on the time of day.
- **draw_cloud(cx, cy)**: Draws clouds.
- **draw_stars()**: Renders stars at night.
- **draw_timestamp(time)**: Displays the current time on the screen.

Time-Handling Functions

- **update_time()**: Updates the current time by adding 0.1 hours and wraps around if it exceeds 24 hours.
- **day_night()**: Determines if it is day or night based on the current time.

OpenGL Setup Functions

- **init()**: Initializes OpenGL settings, including background color and orthographic projection.
- **update(value)**: Updates the animation and schedules the next update.
- **display()**: Handles rendering the entire scene, including background color, celestial objects, buildings, trees, birds, clouds, stars, and timestamp.

Main Function

The `main` function initializes the GLUT framework, sets up the display mode, window size, and position, and starts the main loop. This function brings together all the components and functions, creating a cohesive and dynamic graphical application.

4.2 Performance Analysis of the methods defined

Midpoint Line Algorithm

Efficiency: The Midpoint Line Algorithm is efficient for rasterizing lines due to its incremental error calculation, reducing the need for floating-point arithmetic and division operations.

Drawing Functions

Optimization: Each drawing function (buildings, trees, birds, sun/moon, clouds, stars) is optimized to minimize state changes and redundant operations. For example, colors and vertex positions are set before entering loops to avoid repeated function calls.

Bird Movement

Smooth Animation: The update_birds function ensures smooth and continuous bird movement by incrementally updating their positions, maintaining performance even with multiple birds.

Time Progression

Real-Time Simulation: The timer function incrementally advances the current time, allowing the cityscape to transition smoothly between day and night without significant performance overhead.

Rendering Performance

Frame Rate: The use of double buffering (via glutSwapBuffers) ensures smooth rendering and avoids flickering. The rendering performance is optimized by minimizing the number of state changes and leveraging efficient algorithms for drawing primitives.

User Interaction

Responsiveness: The mouse_click function is designed to respond promptly to user inputs, providing a responsive and interactive experience without affecting overall performance.

In summary, the methods defined in this project are designed to balance visual fidelity and performance, ensuring smooth animation and interaction within the constraints of real-time rendering.

CHAPTER 5.

Implementation

The implementation of the dynamic cityscape simulation involves setting up the OpenGL environment, defining the necessary graphical elements, and implementing the logic for animation and user interaction. The following sections provide detailed explanations of the key components and their implementation.

Initial Setup

1. Import Libraries

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
import sys
```

2. Define Constants and Variables

```
window_width = 800
window_height = 600
current_time = 0  # Time variable to simulate day and night
bird_positions = [(np.random.randint(0, window_width), np.random.randint(300, window_height))
for _ in range(10)]
```

Initialization

3. Initialize OpenGL Environment

```
def init():
    glClearColor(0.0, 0.0, 0.0, 1.0)  # Set clear color to black
    gluOrtho2D(0, window_width, 0, window_height)
```

Drawing Functions

4. Background Color

```
    def BackGroundColour():
    global current_time
    if 6 <= current_time < 18:
        glClearColor(0.53, 0.81, 0.92, 1.0)  # Daytime sky color
    else:
        glClearColor(0.0, 0.0, 0.1, 1.0)  # Nighttime sky color
```

5. Draw Buildings

```
def draw_building(x, width, height, color):
    glColor3f(*color)
    glBegin(GL_POLYGON)
    glVertex2f(x, 0)
    glVertex2f(x + width, 0)
    glVertex2f(x + width, height)
    glVertex2f(x, height)
    glEnd()
```

6. Draw Trees

```
def draw_tree(x, y):
    glColor3f(0.55, 0.27, 0.07)
    glBegin(GL_POLYGON)
    glVertex2f(x, y)
    glVertex2f(x + 10, y)
    glVertex2f(x + 10, y + 40)
    glVertex2f(x, y + 40)
    glEnd()
    glColor3f(0.0, 0.8, 0.0)
    glBegin(GL_TRIANGLES)
    glVertex2f(x - 20, y + 40)
    glVertex2f(x + 30, y + 40)
    glVertex2f(x + 5, y + 70)
    glEnd()
```

7. Draw Birds

```
def draw_bird(x, y):
    glColor3f(0.0, 0.0, 0.0)
    glBegin(GL_LINES)
    glVertex2f(x, y)
    glVertex2f(x + 10, y + 10)
    glVertex2f(x, y)
    glVertex2f(x - 10, y + 10)
    glEnd()
```

8. Draw Sun and Moon

```
def draw_sun_and_moon():
    global current_time
    if 6 <= current_time < 18:
        glColor3f(1.0, 1.0, 0.0)  # Yellow sun
        draw_circle(700, 500, 50)
    else:
        glColor3f(1.0, 1.0, 1.0)  # White moon
        draw_circle(700, 500, 50)

def draw_circle(x, y, radius):
    glBegin(GL_POLYGON)
    for i in range(360):
        angle = np.radians(i)
        glVertex2f(x + np.cos(angle) * radius, y + np.sin(angle) * radius)
    glEnd()
```

9. Draw Clouds

```
def draw_cloud(x, y):
    glColor3f(1.0, 1.0, 1.0)
    draw_circle(x, y, 20)
    draw_circle(x + 20, y + 10, 20)
    draw_circle(x + 40, y, 20)
```

10. Draw Stars

```
def draw_stars():
    global current_time
    if current_time < 6 or current_time >= 18:
        glColor3f(1.0, 1.0, 1.0)
        for _ in range(100):
            x = np.random.randint(0, window_width)
            y = np.random.randint(300, window_height)
            glBegin(GL_POINTS)
            glVertex2f(x, y)
            glEnd()
```

Animation and Interaction

11. Bird Movement

```
def update_birds(value):
    global bird_positions
    bird_positions = [(x + 2 if x < window_width else 0, y) for x, y in bird_positions]
    glutPostRedisplay()
    glutTimerFunc(50, update_birds, 0)
```

12. Mouse Interaction

```
def mouse_click(button, state, x, y):
    if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:
        bird_positions.append((x, window_height - y))
```

13. Time Progression

```
def timer(value):
    global current_time
    current_time = (current_time + 1) % 24
    glutPostRedisplay()
    glutTimerFunc(1000, timer, 0)
```

13. Time Progression

```
def timer(value):
    global current_time
    current_time = (current_time + 1) % 24
    glutPostRedisplay()
    glutTimerFunc(1000, timer, 0)
```

Display and Main Loop

14. Display Function

```
def display():
    glClear(GL_COLOR_BUFFER_BIT)
    BackGroundColour()
    draw_building(100, 80, 200, (0.75, 0.75, 0.75))
    draw_tree(200, 100)
    for x, y in bird_positions:
        draw_bird(x, y)
    draw_sun_and_moon()
    draw_cloud(300, 500)
    draw_stars()
    glutSwapBuffers()
```

15. Main Function

```
def main():
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB)
    glutInitWindowSize(window_width, window_height)
    glutCreateWindow("Dynamic Cityscape")
    init()
    glutDisplayFunc(display)
    glutMouseFunc(mouse_click)
    glutTimerFunc(50, update_birds, 0)
    glutTimerFunc(1000, timer, 0)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

5.1 Data flow Diagram

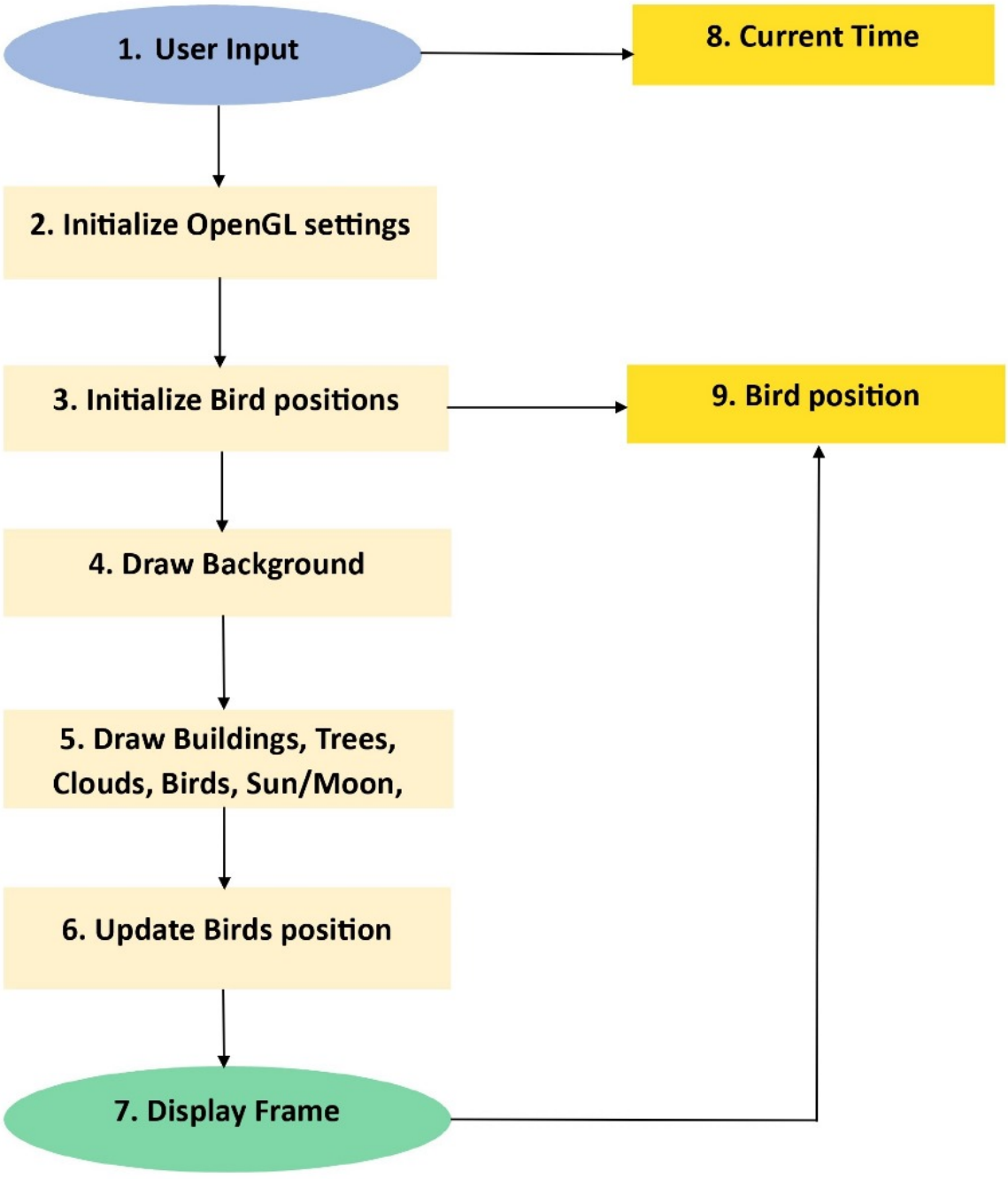

Fig 5.1 Implementation dataflow diagram

1. **User Input Time**:
    - The user inputs the time, which is used to determine the current time for the simulation.

2. **Initialize OpenGL Settings**:
    - The initial settings for OpenGL are configured.

3. **Initialize Bird Positions**:
    - The initial positions of the birds are set

4. **Draw Background**:
    - The background of the scene is drawn.

5. **Draw Buildings, Trees, Clouds, Birds, Sun/Moon, Stars**:
    - The various elements of the cityscape, including buildings, trees, clouds, birds, the sun, the moon, and stars, are drawn.

6. **Update Birds Position**:
    - The positions of the birds are updated based on the current time and their initial positions.

7. **Display Frame**:
    - The current frame of the simulation is displayed.

Additionally, the flowchart shows that the current time is used to update the bird positions dynamically. This ensures that the simulation reflects changes in real-time.

This flowchart effectively outlines the steps and the flow of your simulation process, from user input to displaying the final frame, ensuring a structured approach to building the cityscape simulation.

CHAPTER 6.

Results and Snapshot

1.Morning (6 AM)

- The transition from night to day.
- The sun rising, casting a gentle light.
- Buildings, trees, and birds becoming more visible.

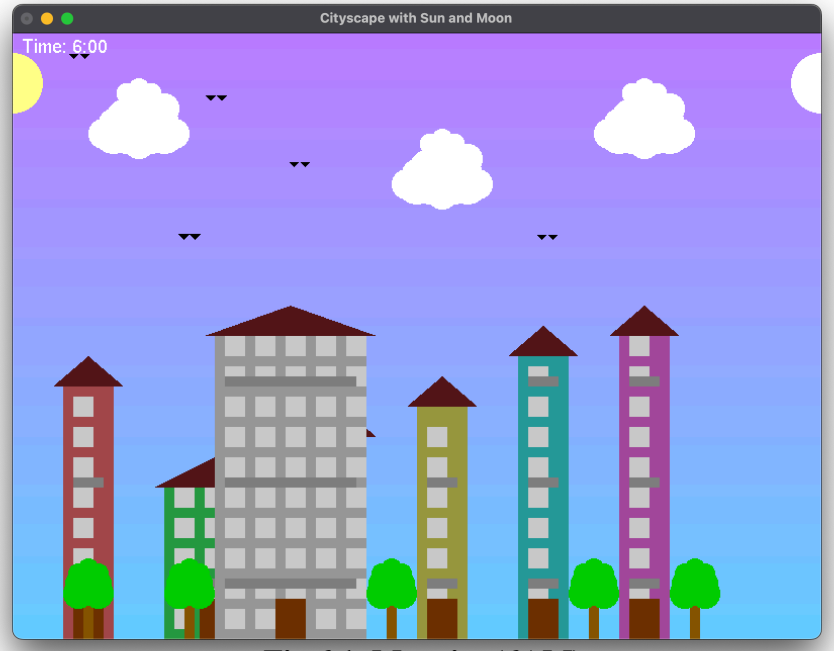

Fig 6.1 Morning(6AM)

2. Noon (12 PM)

- Bright daylight with the sun at its highest point.
- Full visibility of buildings, trees, and clouds.
- Birds flying across a clear sky.

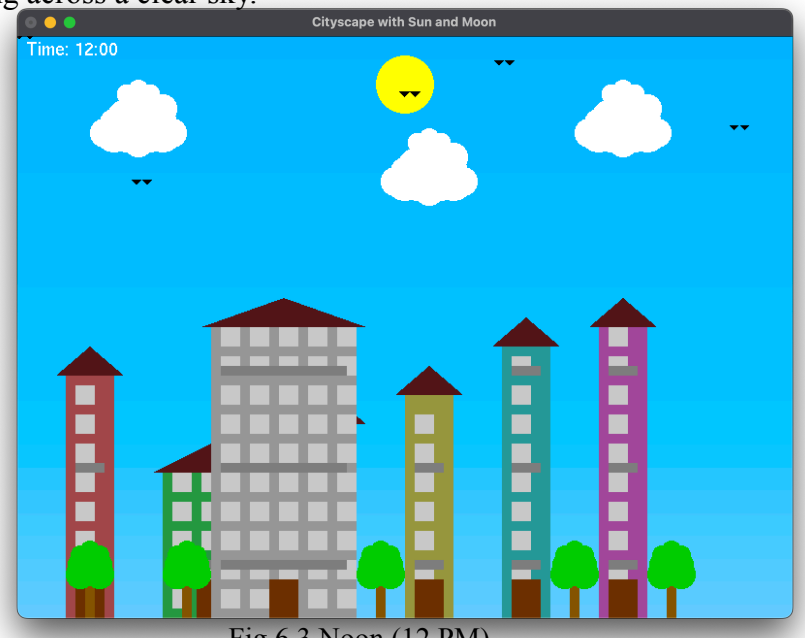

Fig 6.3 Noon (12 PM)

3. Afternoon (3 PM)

- A well-lit environment with shadows indicating the direction of sunlight.
- Active bird movement.
- Presence of clouds adding depth to the sky.

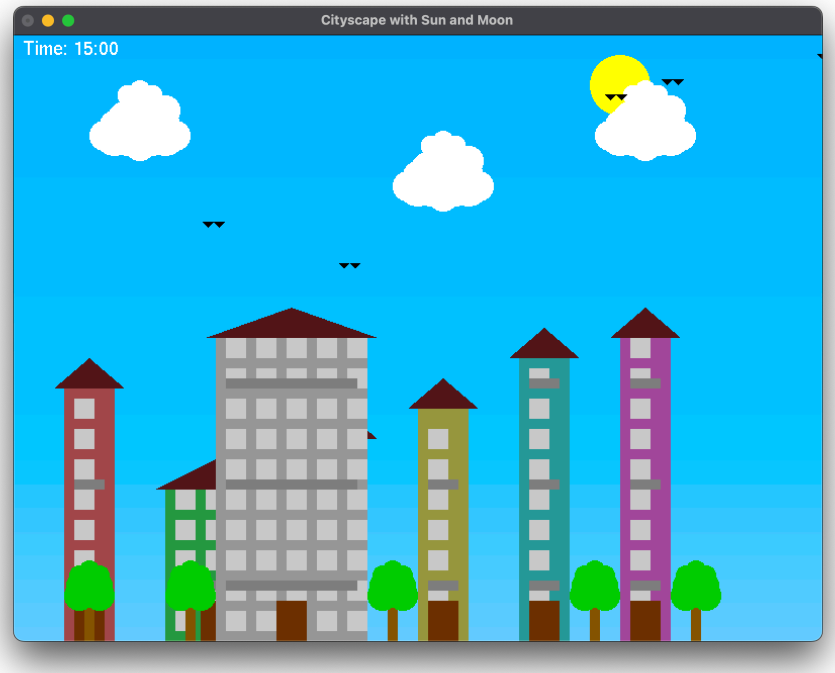

Fig 6.3 Afternoon (3 PM)

4. Evening (6 PM)

- The transition from day to night.
- The sun setting, creating a warm glow.
- Buildings and trees starting to cast longer shadows.

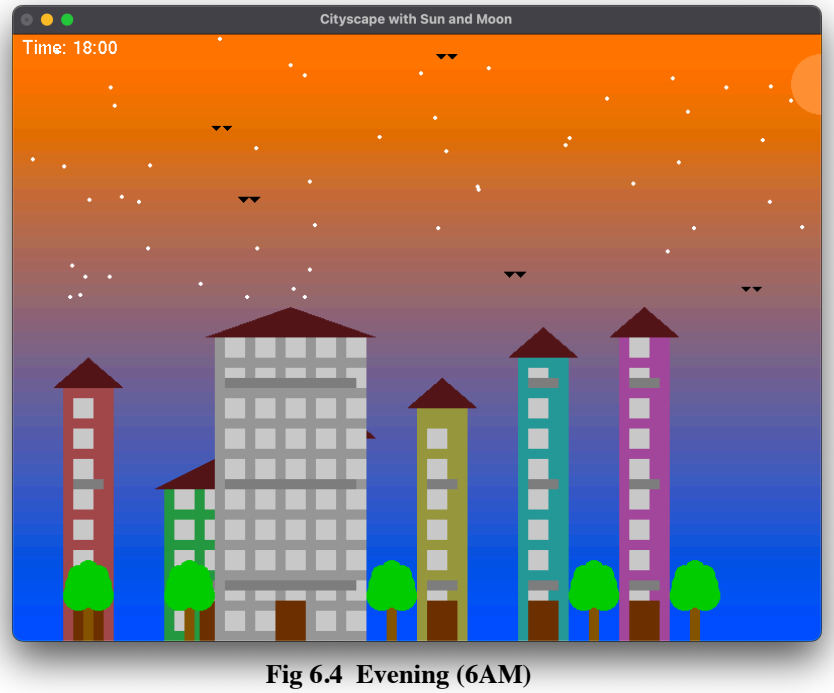

Fig 6.4 Evening (6AM)

5. Night (9 PM)

- The night sky with stars and the moon visible.
- Buildings and trees in silhouette against the dark sky.
- Birds resting or minimal movement.

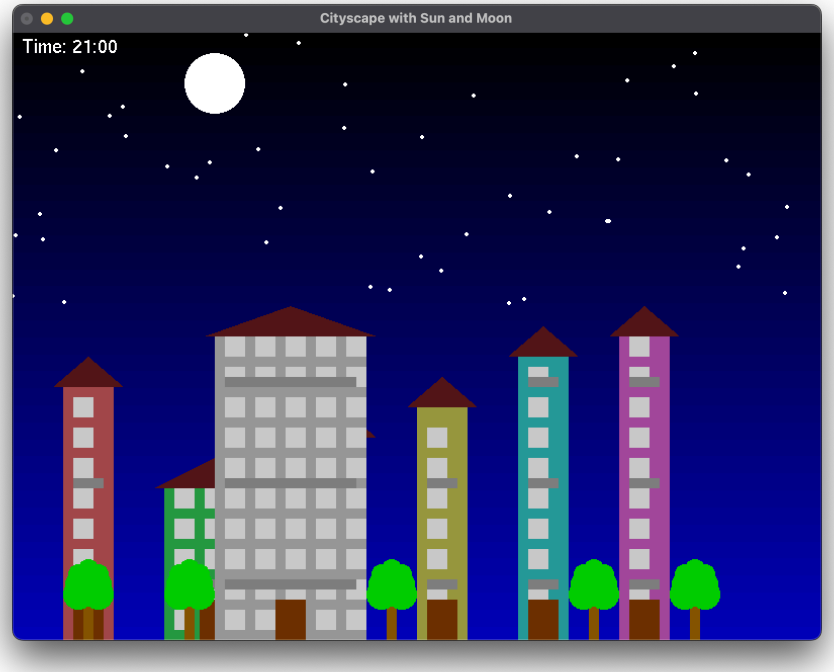

Fig 6.5 Night (9AM)

6. Midnight (12 AM)

- A dark sky with prominent stars and the moon.
- Stillness in the environment.
- Buildings and trees barely visible, creating a serene night scene.

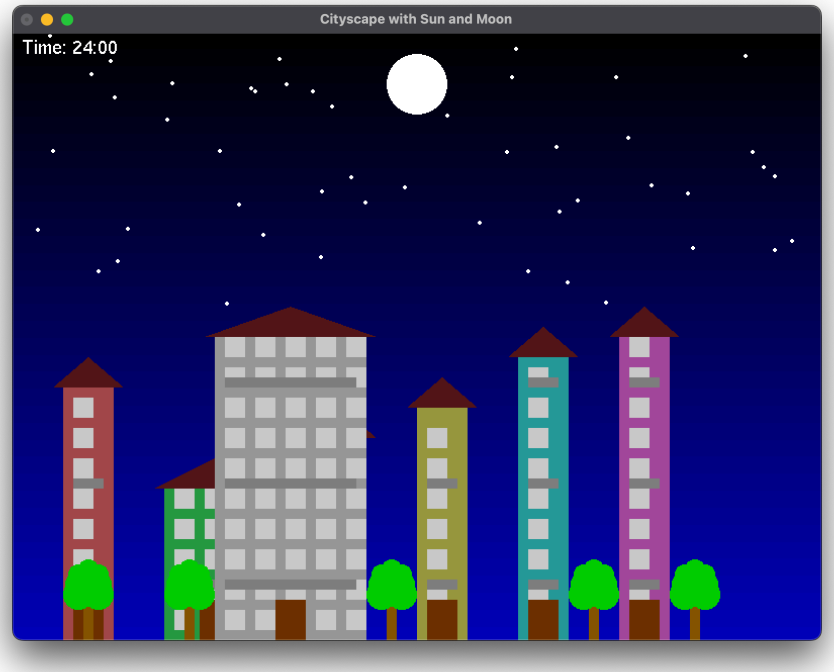

Fig6.6 Midnight (12AM)

Conclusion

Successful Demonstration:

- Created a dynamic cityscape simulation using OpenGL and GLUT.
- Incorporated buildings, trees, celestial bodies, clouds, and birds.
- Achieved seamless transitions between day and night based on a simulated time variable.

Implementation Techniques:

- Used geometric algorithms for drawing primitives.
- Applied color interpolation techniques for realistic lighting effects and atmospheric changes.

Methodology:

- Detailed planning and systematic implementation of OpenGL functions.
- Managed rendering, user interactions, and time-based animations effectively.

Performance Optimizations:

- Ensured smooth rendering and real-time interaction.
- Utilized OpenGL's rendering pipeline and data structures efficiently.

Achievement of Objectives:

- Met goals of creating a visually engaging cityscape simulation.
- Provided valuable insights into OpenGL programming and graphics rendering techniques.

Future Enhancements:

- Potential to explore more complex environmental dynamics.
- Opportunities to add interactive features and further optimizations for enhanced realism and user engagement.

References

1. Hughes, J. F., van Dam, A., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K., & Akeley, K. (2013). Computer Graphics: Principles and Practice. Pearson Education.
2. Woo, M., Neider, J., Davis, T., & Shreiner, D. (2013). OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3. Addison-Wesley Professional.
3. Lindley, C. A. (1993). Advanced Computer Graphics: Proceedings of Computer Graphics Tokyo '93. Springer.
4. Hearn, D., & Baker, M. P. (2014). Computer Graphics with OpenGL. Pearson Education.
5. Angel, E., & Shreiner, D. (2013). Interactive Computer Graphics: A Top-Down Approach with WebGL. Addison-Wesley.

Bibliography

1. OpenGL Documentation. Retrieved from https://github.com/KhronosGroup/OpenGL-Refpages/tree/main/gl4/html
2. Python OpenGL (PyOpenGL) Documentation. Retrieved from http://pyopengl.sourceforge.net/

Appendix:Source Code

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
import time

# Window dimensions
width, height = 800, 600

# Time in hours
current_time = 12.0

# Birds' positions
num_birds = 5
bird_positions = np.zeros((num_birds, 2))
def initialize_bird_positions():
    for i in range(num_birds):
        bird_positions[i] = [np.random.uniform(0, width), np.random.uniform(height / 2, height)]
initialize_bird_positions()
def draw_points(x, y):
    glBegin(GL_POINTS)
    glVertex2i(int(x), int(y))
    glEnd()
def findZone(x0, y0, x1, y1):
    dy = y1 - y0
    dx = x1 - x0
 if abs(dx) > abs(dy):
        if dx > 0 and dy >= 0:
            return 0
        elif dx < 0 <= dy:
            return 3
        elif dx < 0 and dy < 0:
            return 4
        else:
            return 7
    else:
        if dx > 0 and dy > 0:
            return 1
        elif dx <= 0 < dy:
            return 2
        elif dx < 0 and dy < 0:
            return 5
        else:
            return 6
```

```
elif dx < 0 and dy < 0:
            return 5
        else:
            return 6

def ZoneZeroConversion(zone, x, y):
    if zone == 0:
        return x, y
    elif zone == 1:
        return y, x
    elif zone == 2:
        return y, -x
    elif zone == 3:
        return -x, y
    elif zone == 4:
        return -x, -y
    elif zone == 5:
        return -y, -x
    elif zone == 6:
        return -y, x
    elif zone == 7:
        return x, -y

def zero_to_original_zone(zone, x, y):
    if zone == 0:
        return x, y
    if zone == 1:
        return y, x
    if zone == 2:
        return -y, x
    if zone == 3:
        return -x, y
    if zone == 4:
        return -x, -y
    if zone == 5:
        return -y, -x
    if zone == 6:
        return y, -x
    if zone == 7:
        return x, -y

def MidPointLine(zone, x0, y0, x1, y1):
    dy = y1 - y0
    dx = x1 - x0
```

```
d_init = 2 * dy - dx
    e = 2 * dy
    ne = 2 * (dy - dx)

    x = x0
    y = y0
    while x <= x1:
        a, b = zero_to_original_zone(zone, x, y)
        draw_points(a, b)
        if d_init <= 0:
            x += 1
            d_init += e
        else:
            x += 1
            y += 1
            d_init += ne
def eight_way_symmetry(x0, y0, x1, y1):
    zone = findZone(x0, y0, x1, y1)
    z0_x0, z0_y0 = ZoneZeroConversion(zone, x0, y0)
    z0_x1, z0_y1 = ZoneZeroConversion(zone, x1, y1)
    MidPointLine(zone, z0_x0, z0_y0, z0_x1, z0_y1)
def interpolate_color(color1, color2, factor):
    return [color1[i] + factor * (color2[i] - color1[i]) for i in range(3)]

def BackGroundColour(x):
    if x > 24 or x < 0:
        print("Invalid Time")
    else:
        if 6 <= x <= 8:
            factor = (x - 6) / 12.0
            top_color = [0.7, 0.5, 1.0]  # Day top color
            bottom_color = [0.5, 0.8, 1.0]  # Day bottom color
        elif 9 <= x <= 11:
            factor = (x - 6) / 12.0
            top_color = [0.4, 0.5, 1.0]  # Mid-day top color
            bottom_color = [0.5, 0.8, 1.0]  # Mid-day bottom color
        elif 12 <= x <= 15:
            factor = (x - 18) / 6.0
            top_color = [0.0, 0.7, 1.0]  # Noon top color
            bottom_color = [0.5, 0.8, 1.0]# Noon bottom color
        elif 16 <= x <= 18:
            factor = (x - 18) / 6.0
            top_color = [1.0, 0.5, 0.0]  # Evening top color
            bottom_color = [0.0, 0.3, 1.3]  # Evening bottom color
```

```
        bottom_color = [0.0, 0.0, 0.7]  # Night bottom color

        glBegin(GL_QUADS)
        glColor3f(*top_color)
        glVertex2f(0, height)
        glVertex2f(width, height)
        glColor3f(*bottom_color)
        glVertex2f(width, 0)
        glVertex2f(0, 0)
        glEnd()

def draw_building(x0, y0, width, height, color):
    glColor3f(*color)  # Building color
    glBegin(GL_POLYGON)
    glVertex2f(x0, y0)
    glVertex2f(x0, y0 + height)
    glVertex2f(x0 + width, y0 + height)
    glVertex2f(x0 + width, y0)
    glEnd()

    # Draw windows
    glColor3f(0.8, 0.8, 0.8)  # Window color
    for i in range(10, height - 10, 30):
        for j in range(10, width - 10, 30):
            glBegin(GL_POLYGON)
            glVertex2f(x0 + j, y0 + i)
            glVertex2f(x0 + j + 20, y0 + i)
            glVertex2f(x0 + j + 20, y0 + i + 20)
            glVertex2f(x0 + j, y0 + i + 20)
            glEnd()

    # Draw door
    glColor3f(0.4, 0.2, 0.0)  # Door color
    glBegin(GL_POLYGON)
    glVertex2f(x0 + width / 2 - 15, y0)
    glVertex2f(x0 + width / 2 - 15, y0 + 40)
    glVertex2f(x0 + width / 2 + 15, y0 + 40)
    glVertex2f(x0 + width / 2 + 15, y0)
    glEnd()
```

```
# Draw roof
    glColor3f(0.3, 0.1, 0.1)  # Roof color
    glBegin(GL_TRIANGLES)
    glVertex2f(x0 - 10, y0 + height)
    glVertex2f(x0 + width / 2, y0 + height + 30)
    glVertex2f(x0 + width + 10, y0 + height)
    glEnd()

    # Additional architectural details
    # Balconies
    for i in range(50, height, 100):
        glColor3f(0.5, 0.5, 0.5)
        glBegin(GL_POLYGON)
        glVertex2f(x0 + 10, y0 + i)
        glVertex2f(x0 + 10, y0 + i + 10)
        glVertex2f(x0 + width - 10, y0 + i + 10)
        glVertex2f(x0 + width - 10, y0 + i)
        glEnd()

def draw_tree(x0, y0):
    glColor3f(0.5, 0.35, 0.05)  # Brown color for trunk
    glBegin(GL_POLYGON)
    glVertex2f(x0, y0)
    glVertex2f(x0, y0 + 40)
    glVertex2f(x0 + 10, y0 + 40)
    glVertex2f(x0 + 10, y0)
    glEnd()

    # Draw the leaves
    glColor3f(0.0, 0.8, 0.0)  # Green color for leaves
    draw_circle(x0 + 5, y0 + 50, 15)  # Bottom layer
    draw_circle(x0 - 5, y0 + 45, 15)
    draw_circle(x0 + 15, y0 + 45, 15)
    draw_circle(x0 + 5, y0 + 60, 13)  # Middle layer
    draw_circle(x0 - 5, y0 + 55, 13)
    draw_circle(x0 + 15, y0 + 55, 13)
    draw_circle(x0 + 5, y0 + 70, 10)  # Top layer
    draw_circle(x0 - 5, y0 + 65, 10)
    draw_circle(x0 + 15, y0 + 65, 10)
```

```
def draw_bird(x, y, size=1.1):
    glColor3f(0, 0, 0)  # Black color for birds
    glBegin(GL_TRIANGLES)
    glVertex2f(x, y)
    glVertex2f(x - 5 * size, y + 5 * size)
    glVertex2f(x + 5 * size, y + 5 * size)
    glEnd()
    glBegin(GL_TRIANGLES)
    glVertex2f(x + 10 * size, y)
    glVertex2f(x + 5 * size, y + 5 * size)
    glVertex2f(x + 15 * size, y + 5 * size)
    glEnd()

def update_birds():
    global bird_positions
    for i in range(len(bird_positions)):
        bird_positions[i, 0] += 1
        if bird_positions[i, 0] > width:
            bird_positions[i, 0] = 0


def draw_circle(cx, cy, radius):
    num_segments = 100
    theta = 2 * np.pi / num_segments
    c = np.cos(theta)
    s = np.sin(theta)
    x = radius
    y = 0

    glBegin(GL_POLYGON)
    for _ in range(num_segments):
        glVertex2f(x + cx, y + cy)
        t = x
        x = c * x - s * y
        y = s * t + c * y
    glEnd()
```

```
def draw_sun_and_moon(time):
    if 6 <= time <= 9:
        # Sun during the day
        sun_position = (time - 6) / 12 * width
        glColor3f(1.0, 1.0, 0.6) # Lighter yellow for edges
        draw_circle(sun_position, height - 50, 30)
    if 9 <= time <= 11:
        # Sun during the mid-day
        sun_position = (time - 6) / 12 * width
        glColor3f(1.0, 1.0, 0.8) # Light Yellow color for the sun (mid-day)
        draw_circle(sun_position, height - 50, 30)
    if 12 <= time <= 15:
        # Sun during the afternoon
        sun_position = (time - 6) / 12 * width
        glColor3f(1.0, 1.0, 0.0) # Bright Yellow color for the sun (noon)
        draw_circle(sun_position, height - 50, 30)
    if 16 <= time <= 18:
        # Sun during the evening
        sun_position = (time - 6) / 12 * width
        glColor3f(1.0, 0.6, 0.3) # Orange for top
        draw_circle(sun_position, height - 50, 30)
    else:
        # Moon during the night
        if time >= 18:
            moon_position = (time - 18) / 12 * width
        else:
            moon_position = (time + 6) / 12 * width
        glColor3f(1.0, 1.0, 1.0)  # White color for the moon
        draw_circle(moon_position, height - 50, 30)

def draw_cloud(cx, cy):
    glColor3f(1.0, 1.0, 1.0)  # White color for clouds

    # Main body of the cloud
    draw_circle(cx, cy, 20)
    draw_circle(cx + 25, cy, 20)
    draw_circle(cx + 50, cy, 20)
    draw_circle(cx + 12.5, cy + 15, 20)
    draw_circle(cx + 37.5, cy + 15, 20)
    draw_circle(cx + 25, cy + 30, 15)
    draw_circle(cx + 50, cy + 10, 15)
    draw_circle(cx, cy + 10, 15)
    draw_circle(cx + 50, cy + 25, 15)
```

```
    # Additional circles to cloud
    draw_circle(cx - 10, cy, 15)  # Left side
    draw_circle(cx + 60, cy, 15)  # Right side
    draw_circle(cx + 25, cy - 10, 15)  # Bottom middle
    draw_circle(cx + 37.5, cy - 10, 10)  # Bottom right
    draw_circle(cx + 12.5, cy - 10, 10)  # Bottom left
    draw_circle(cx + 25, cy + 45, 10)  # Top middle
    draw_circle(cx + 37.5, cy + 40, 10)  # Top right
    draw_circle(cx + 12.5, cy + 40, 10)  # Top left


def draw_stars():
    glColor3f(1.0, 1.0, 1.0)  # White color for stars
    for _ in range(50):
        x = np.random.randint(0, width)
        y = np.random.randint(height // 1.8, height)
        draw_circle(x, y, 2)

def draw_timestamp(time):
    time_str = f"Time: {int(time)}:00"
    glColor3f(1.0, 1.0, 1.0)  # White color for the text
    glRasterPos2f(10, height - 20)
    for char in time_str:
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, ord(char))

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    BackGroundColour(current_time)

    # Draw buildings with different colors and details
    draw_building(50, 0, 50, 250, (0.6, 0.3, 0.3))  # Building 1
    draw_building(150, 0, 100, 150, (0.3, 0.6, 0.3))  # Building 2
    draw_building(300, 0, 50, 200, (0.3, 0.3, 0.6))  # Building 3
    draw_building(400, 0, 50, 230, (0.6, 0.6, 0.3))  # Building 4
    draw_building(500, 0, 50, 280, (0.3, 0.6, 0.6))  # Building 5
    draw_building(600, 0, 50, 300, (0.6, 0.3, 0.6))  # Building 6
    draw_building(200, 0, 150, 300, (0.6, 0.6, 0.6))  # Building 7
```

```
    # Draw trees
    draw_tree(70, 0)
    draw_tree(170, 0)
    draw_tree(370, 0)
    draw_tree(570, 0)
    draw_tree(670, 0)

    # Draw sun or moon based on the time
    draw_sun_and_moon(current_time)

    # Draw clouds during the day
    if 6 <= current_time < 18:
        draw_cloud(100, height - 100)
        draw_cloud(400, height - 150)
        draw_cloud(600, height - 100)

    # Draw stars at night
    if current_time < 6 or current_time >= 18:
        draw_stars()

    # Draw timestamp
    draw_timestamp(current_time)

    # Draw birds
    if 6 <= current_time <= 18:
        for bird in bird_positions:
            draw_bird(bird[0], bird[1])

    update_birds()
    glutSwapBuffers()

def mouse_click(button, state, x, y):
    if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:
        for i in range(len(bird_positions)):
            bird_positions[i, 1] += 3

    glFlush()

def init():
    glClearColor(0.0, 0.0, 0.0, 1.0)
    gluOrtho2D(0, width, 0, height)
```

```
def timer(value):
    global current_time
    current_time += 0.01
    if current_time >= 24.0:
        current_time = 0.0
    glutPostRedisplay()
    glutTimerFunc(100, timer, 0)

def main():
    global current_time

    # User input for time
    time_input = input("Enter the time in hours (24 hrs Format): ")
    try:
        current_time = float(time_input)
        if current_time < 0 or current_time > 24:
            print("Invalid input. Defaulting to noon (12.0).")
            current_time = 12.0
    except ValueError:
        print("Invalid input. Defaulting to noon (12.0).")
        current_time = 12.0

    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(width, height)
    glutCreateWindow(b"Cityscape with Sun and Moon")
    init()
    initialize_bird_positions()
    glutDisplayFunc(display)
    glutMouseFunc(mouse_click)
    glutMainLoop()

if __name__ == "__main__":
    main()
```