МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

по дисциплине Компьютерная графика

Тема: «Исследование алгоритмов выявления видимости сложных сцен»

Студенты гр. 1307 Грунская Н.Д.
Тростин М.Ю.
Голубев М.А.

Преподаватель _____ Матвеева И.В.
_____

Санкт-Петербург

2024

Цель работы:

Практическое закрепление теоретических знаний об исследовании алгоритмов выявления видимости сложных сцен.

Постановка задачи:

Обеспечить реализацию видимости совокупности произвольных многогранников на основе использования алгоритма деления окна пополам (алгоритма Варнока).

Краткая теоретическая информация:

Алгоритм Варнока работает в пространстве изображения и анализирует область на экране дисплея (окно) на наличие в них видимых элементов. Если в окне нет изображения, то оно просто закрашивается фоном. Если же в окне имеется элемент, то проверяется, достаточно ли он прост для визуализации. Если объект сложный, то окно разбивается на более мелкие, для каждого из которых выполняется тест на отсутствие и/или простоту изображения. Рекурсивный процесс разбиения может продолжаться до тех пор, пока не будет достигнут предел разрешения экрана.

Можно выделить 4 случая взаимного расположения окна и многоугольника:

1. многоугольник целиком вне окна
2. многоугольник целиком внутри окна
3. многоугольник пересекает окно
4. многоугольник охватывает окно

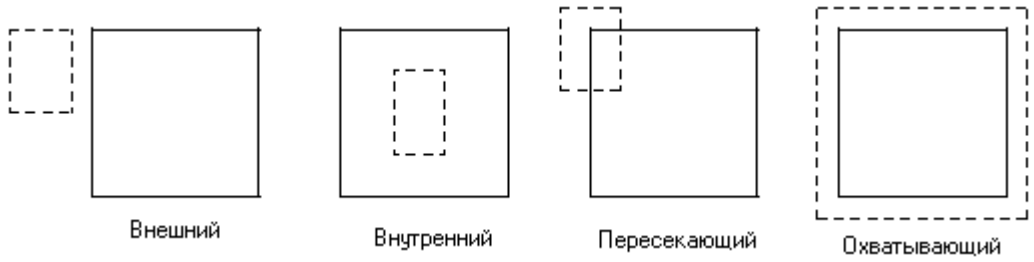

Рисунок 1. Взаимное расположение окна и многоугольника.

В любых других случаях процесс разбиения окна продолжается. Легко видеть, что при растре 1024×1024 и делении стороны окна пополам требуется не более 10 разбиений. Если достигнуто максимальное разбиение, но не обнаружено ни одного из приведенных выше четырех случаев, то для точки с центром в полученном минимальном окне (размером в пиксель) вычисляются глубины оставшихся многоугольников и закраску определяет многоугольник, наиболее близкий к наблюдателю. При этом для устранения лестничного эффекта можно выполнить дополнительные разбиения и закрасить пиксел с учетом всех многоугольников, видимых в минимальном окне.

Первые три случая идентифицируются легко. Последний же случай фактически сводится к поиску охватывающего многоугольника, перекрывающего все остальные многоугольники, связанные с окном. Проверка на такой многоугольник может быть выполнена следующим образом: в угловых точках окна вычисляются Z-координаты для всех многоугольников, связанных с окном. Если все четыре такие Z-координаты охватывающего многоугольника ближе к наблюдателю, чем все остальные, то окно закрашивается цветом соответствующего охватывающего многоугольника. Если же нет, то мы имеем сложный случай и разбиение следует продолжить.

Очевидно, что после разбиения окна охватывающие и внешние многоугольники наследуются от исходного окна. Поэтому необходимо проверять лишь внутренние и пересекающие многоугольники.

Из изложенного ясно, что важной частью алгоритма является определение расположения многоугольника относительно окна.

Проверка на то что многоугольник внешний или внутренний относительно окна для случая прямоугольных окон легко реализуется использованием прямоугольной оболочки многоугольника и сравнением координат. Для внутреннего многоугольника должны одновременно выполняться условия:

$$\begin{cases} X_{\min} \geq W_л \\ X_{\max} \leq W_п \\ Y_{\min} \geq W_н \\ Y_{\max} \leq W_в \end{cases}$$

где Xmin, Xmax, Ymin, Ymax – ребра оболочки, Wл, Wп, Wн, Wв – ребра окна.

Для внешнего многоугольника достаточно выполнение любого из следующих условий:

$$X_{\min} < W_л, X_{\max} > W_п, Y_{\min} < W_н, Y_{\max} > W_в$$

Таким способом внешний многоугольник, охватывающий угол окна не будет идентифицирован как внешний:

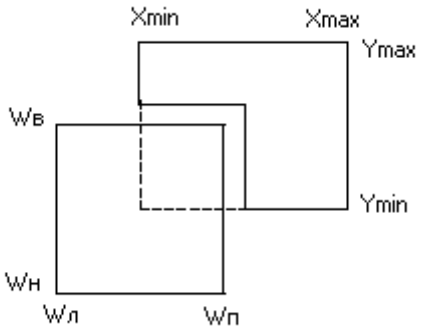

Рисунок 2. Ошибочное определение внешнего многоугольника как пересекающего при использовании прямоугольной оболочки.

Пример работы программы:

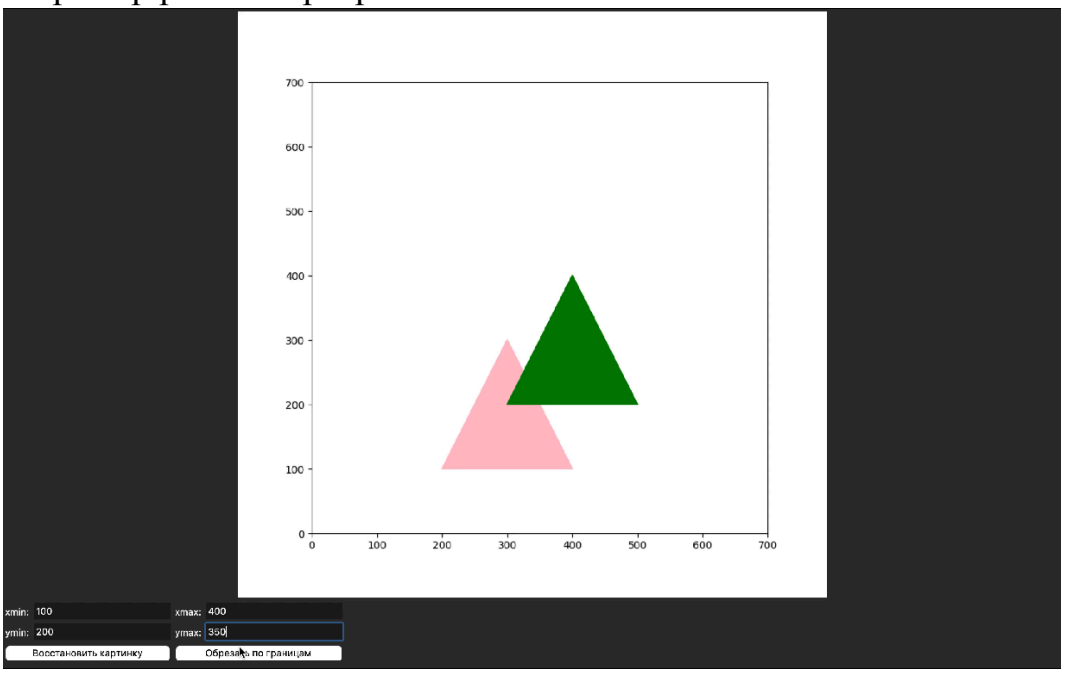

Рисунок 3. Пример работы программы.

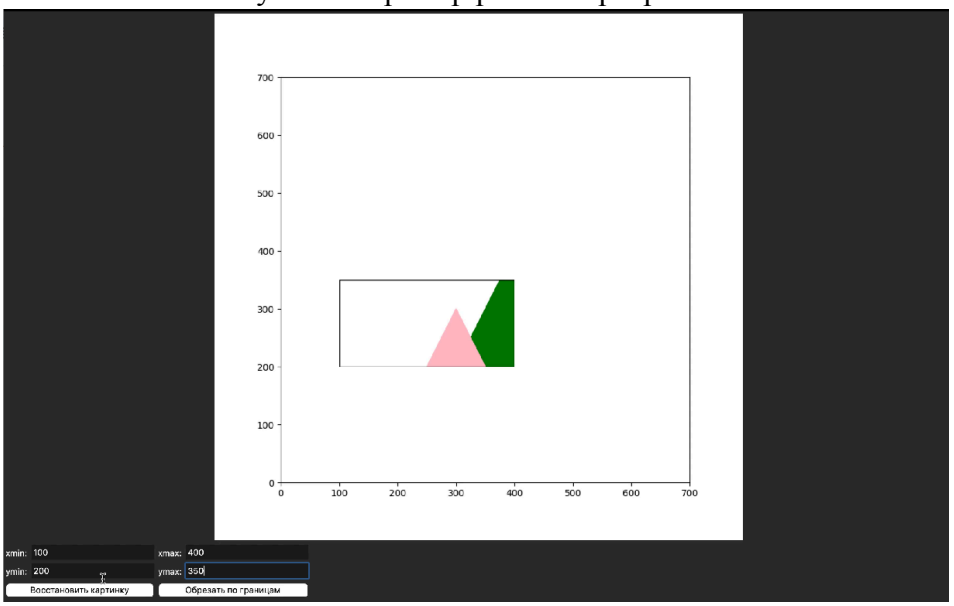

Рисунок 5. Пример работы программы.

Реализация алгоритма

Для реализации алгоритма был использован Python, а для визуализации результата был использован Tkinter.

```
class WarnockAlgorithm:
    def __init__(self, scene, ax, depth=3):
        self.scene = scene
        self.ax = ax
        self.depth = depth

    def run(self, viewport=None):
        # If no viewport is provided, use the full viewport
        if viewport is None:
            viewport = Viewport(
                self.ax.get_xlim()[0],
                self.ax.get_xlim()[1],
                self.ax.get_ylim()[0],
                self.ax.get_ylim()[1]
            )
        # Start the division and conquer algorithm
        self.divide_and_conquer(viewport, self.depth)

    def divide_and_conquer(self, viewport, depth):
        # Base case: if the depth is zero, draw the viewport
        if depth == 0:
            self.render_triangles(viewport)
            return

        # Divide the viewport into 4 smaller viewports
        x_mid = (viewport.xmin + viewport.xmax) / 2
        y_mid = (viewport.ymin + viewport.ymax) / 2

        viewports = [
            Viewport(viewport.xmin, x_mid, viewport.ymin, y_mid),
            Viewport(x_mid, viewport.xmax, viewport.ymin, y_mid),
            Viewport(viewport.xmin, x_mid, y_mid, viewport.ymax),
            Viewport(x_mid, viewport.xmax, y_mid, viewport.ymax)
        ]

        for vp in viewports:
            self.divide_and_conquer(vp, depth - 1)

    def render_triangles(self, viewport):
        # Get triangles that are in the current viewport
```

```
        triangles = self.scene.get_triangles_in_viewport(viewport)

        # Render triangles
        for tri in triangles:
            self.draw_triangle(tri)

    def draw_triangle(self, triangle):
        # Convert 3D points to 2D for rendering
        points = [
            (triangle.p1.x, triangle.p1.y),
            (triangle.p2.x, triangle.p2.y),
            (triangle.p3.x, triangle.p3.y)
        ]
        polygon = Polygon(points, closed=True, color=triangle.color)
        self.ax.add_patch(polygon)
```

Выводы:

Были практически закреплены теоретические знания об алгоритмах выявления видимости сложных сцен, а также об алгоритме деления окна пополам - алгоритме Варнока.

Приложение

Ссылка на видео: https://youtu.be/f3efAzrx9Mc

Исходный код:

```
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
from matplotlib.patches import Polygon, Rectangle
import numpy as np
from matplotlib.path import Path
from matplotlib.patches import Polygon, PathPatch
from matplotlib.path import Path

# Helper classes and functions
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


class Triangle:
    def __init__(self, p1, p2, p3, color='gray'):
        self.p1 = p1
        self.p2 = p2
        self.p3 = p3
        self.color = color


class Viewport:
    def __init__(self, xmin, xmax, ymin, ymax):
        self.xmin = xmin
        self.xmax = xmax
        self.ymin = ymin
        self.ymax = ymax

    def contains_triangle(self, triangle):
        # Check if a triangle is within the viewport
        return (
                self.contains_point(triangle.p1) or
                self.contains_point(triangle.p2) or
                self.contains_point(triangle.p3)
        )

    def contains_point(self, point):
        return (
                self.xmin <= point.x <= self.xmax and
                self.ymin <= point.y <= self.ymax
        )
```

```
class Scene:
    def __init__(self):
        self.triangles = []

    def add_triangle(self, triangle):
        self.triangles.append(triangle)

    def get_triangles_in_viewport(self, viewport):
        # Filter triangles that intersect or are contained in the viewport
        return [tri for tri in self.triangles if viewport.contains_triangle(tri)]


class WarnockAlgorithm:
    def __init__(self, scene, ax, depth=3):
        self.scene = scene
        self.ax = ax
        self.depth = depth

    def run(self, viewport=None):
        # If no viewport is provided, use the full viewport
        if viewport is None:
            viewport = Viewport(
                self.ax.get_xlim()[0],
                self.ax.get_xlim()[1],
                self.ax.get_ylim()[0],
                self.ax.get_ylim()[1]
            )
        # Start the division and conquer algorithm
        self.divide_and_conquer(viewport, self.depth)

    def divide_and_conquer(self, viewport, depth):
        # Base case: if the depth is zero, draw the viewport
        if depth == 0:
            self.render_triangles(viewport)
            return

        # Divide the viewport into 4 smaller viewports
        x_mid = (viewport.xmin + viewport.xmax) / 2
        y_mid = (viewport.ymin + viewport.ymax) / 2

        viewports = [
            Viewport(viewport.xmin, x_mid, viewport.ymin, y_mid),
            Viewport(x_mid, viewport.xmax, viewport.ymin, y_mid),
            Viewport(viewport.xmin, x_mid, y_mid, viewport.ymax),
            Viewport(x_mid, viewport.xmax, y_mid, viewport.ymax)
        ]

        for vp in viewports:
            self.divide_and_conquer(vp, depth - 1)
```

```
    def render_triangles(self, viewport):
        # Get triangles that are in the current viewport
        triangles = self.scene.get_triangles_in_viewport(viewport)

        # Render triangles
        for tri in triangles:
            self.draw_triangle(tri)

    def draw_triangle(self, triangle):
        # Convert 3D points to 2D for rendering
        points = [
            (triangle.p1.x, triangle.p1.y),
            (triangle.p2.x, triangle.p2.y),
            (triangle.p3.x, triangle.p3.y)
        ]
        polygon = Polygon(points, closed=True, color=triangle.color)
        self.ax.add_patch(polygon)


# Main application
class App(tk.Tk):
    def __init__(self):
        super().__init__()

        # Setup
        self.title("Warnock Algorithm")
        self.geometry("800x800")

        # Create a matplotlib figure and axis
        self.fig = Figure(figsize=(8, 8), dpi=100)
        self.ax = self.fig.add_subplot(111)
        self.ax.set_xlim(0, 700)
        self.ax.set_ylim(0, 700)

        # Create a canvas for the figure
        self.canvas = FigureCanvasTkAgg(self.fig, self)
        self.canvas.get_tk_widget().pack(expand=1)

        # Create a frame for controls
        control_frame = tk.Frame(self)
        control_frame.pack(fill=tk.BOTH, expand=True)

        # Create and add entries and buttons
        tk.Label(control_frame, text="xmin:").grid(row=0, column=0)
        self.xmin_entry = tk.Entry(control_frame)
        self.xmin_entry.grid(row=0, column=1)

        tk.Label(control_frame, text="xmax:").grid(row=0, column=2)
        self.xmax_entry = tk.Entry(control_frame)
```

```
        self.xmax_entry.grid(row=0, column=3)

        tk.Label(control_frame, text="ymin:").grid(row=1, column=0)
        self.ymin_entry = tk.Entry(control_frame)
        self.ymin_entry.grid(row=1, column=1)

        tk.Label(control_frame, text="ymax:").grid(row=1, column=2)
        self.ymax_entry = tk.Entry(control_frame)
        self.ymax_entry.grid(row=1, column=3)

        draw_button = tk.Button(control_frame, text="Восстановить картинку",
command=self.draw_triangles)
        draw_button.grid(row=2, column=0, columnspan=2, sticky=tk.W + tk.E)

        clip_button = tk.Button(control_frame, text="Обрезать по границам",
command=self.clip_triangles)
        clip_button.grid(row=2, column=2, columnspan=2, sticky=tk.W + tk.E)

        # Create a scene with triangles
        self.scene = Scene()
        self.add_sample_triangles()

        # Run the Warnock algorithm with initial setup
        self.run_warnock_algorithm()

    def add_sample_triangles(self):
        # Add sample triangles with colors to the scene
        tri1 = Triangle(Point(200, 100, 0), Point(400, 100, 0), Point(300, 300, 0), "pink")
        tri2 = Triangle(Point(300, 200, 0), Point(500, 200, 0), Point(400, 400, 0), "green")
        self.scene.add_triangle(tri1)
        self.scene.add_triangle(tri2)

    def draw_triangles(self):
        # Вместо очистки оси, удалите только существующие патчи и линии
        for patch in self.ax.patches:
            patch.remove()
        for line in self.ax.lines:
            line.remove()

        # Заново добавьте треугольники и переопределите алгоритм Варнока
        self.add_sample_triangles()
        self.run_warnock_algorithm()

        # Обновите холст
        self.canvas.draw()



    def clip_triangles(self):
        # Очистите текущие патчи и линии
```

```
    for patch in self.ax.patches:
        patch.remove()
    for line in self.ax.lines:
        line.remove()

    # Получите введенные пользователем границы
    xmin = float(self.xmin_entry.get())
    xmax = float(self.xmax_entry.get())
    ymin = float(self.ymin_entry.get())
    ymax = float(self.ymax_entry.get())

    # Создайте кастомный viewport с введенными границами
    viewport = Viewport(xmin, xmax, ymin, ymax)

    # Создайте экземпляр WarnockAlgorithm и выполните его с кастомным viewport
    warnock = WarnockAlgorithm(self.scene, self.ax, depth=3)
    warnock.run(viewport)

    # Нарисуйте прямоугольник и белый многоугольник вокруг него
    self.draw_square(xmin, xmax, ymin, ymax)
    self.draw_white_rectangles(xmin, xmax, ymin, ymax)
    # Обновите холст
    self.canvas.draw()

def draw_square(self, xmin, xmax, ymin, ymax):
    # Создайте прямоугольник (квадрат) с указанными границами
    square = Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                       edgecolor='black', facecolor='none')
    # Добавьте прямоугольник на график

    self.ax.add_patch(square)

from matplotlib.patches import Rectangle

def draw_white_rectangles(self, xmin, xmax, ymin, ymax):
    # Размеры оси
    xlim = self.ax.get_xlim()
    ylim = self.ax.get_ylim()

    # Прямоугольник сверху
    top_rect = Rectangle(
        (xlim[0], ymax),  # Начальная точка прямоугольника (x, y)
        xlim[1] - xlim[0],  # Ширина прямоугольника
        ylim[1] - ymax,  # Высота прямоугольника
        facecolor='white', edgecolor='none'
    )

    # Прямоугольник снизу
    bottom_rect = Rectangle(
        (xlim[0], ylim[0]),  # Начальная точка прямоугольника (x, y)
```

```
            xlim[1] - xlim[0],  # Ширина прямоугольника
            ymin - ylim[0],  # Высота прямоугольника
            facecolor='white', edgecolor='none'
        )

        # Прямоугольник слева
        left_rect = Rectangle(
            (xlim[0], ymin),  # Начальная точка прямоугольника (x, y)
            xmin - xlim[0],  # Ширина прямоугольника
            ymax - ymin,  # Высота прямоугольника
            facecolor='white', edgecolor='none'
        )

        # Прямоугольник справа
        right_rect = Rectangle(
            (xmax, ymin),  # Начальная точка прямоугольника (x, y)
            xlim[1] - xmax,  # Ширина прямоугольника
            ymax - ymin,  # Высота прямоугольника
            facecolor='white', edgecolor='none'
        )

        # Добавьте прямоугольники на ось
        self.ax.add_patch(top_rect)
        self.ax.add_patch(bottom_rect)
        self.ax.add_patch(left_rect)
        self.ax.add_patch(right_rect)

    def run_warnock_algorithm(self):
        # Create a WarnockAlgorithm instance and run it
        warnock = WarnockAlgorithm(self.scene, self.ax, depth=3)
        warnock.run()

        # Refresh the canvas

        self.canvas.draw()


if __name__ == "__main__":
    app = App()
    app.mainloop()
```