FlightSim report

Alisher Shakhiyev   Alen German   Auyez Zhumashev

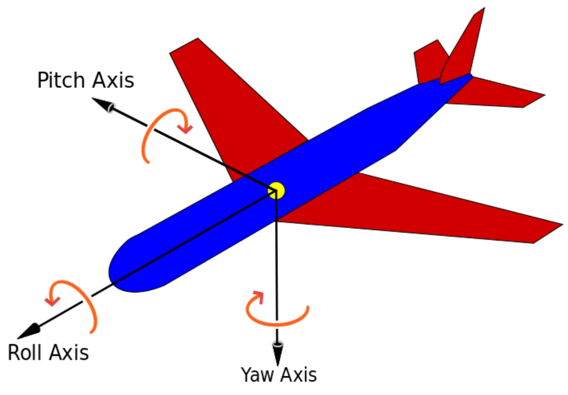

Fig. 1. Control Axes. [17]

I. Introduction

What makes a plane fly? Everything close to the earth is affected by gravitational force. In order to overcome this force and be able to fly planes generate lift force from wings using thrust force from engines. Combining this with aerodynamic drag force, we get four forces that affect the position of the plane in the air: Weight, Lift, Drag, and Thrust. On top of that, planes are able to make maneuvers by rotating in three dimensions, yaw, pitch, and roll (see Fig. 1). These rotations are possible because of the controllable rudder (yaw), elevators (pitch), and ailerons (roll).

The project is done for learning purposes. There is no other motivation other than to acquaint ourselves with airplane physics simulation, basics of 3D graphics, hardware accelerated terrain rendering, and the art of writing video game shaders. This report consists mostly of background information since our work boils down to choosing and implementing existing solutions with no innovation from our side.

In order to get a basic understanding of flight simulation, we inspected existing simulators: professor Hans' simulator, FlightGear [4], and X-Plane [7]. All of them use data measured in wind tunnels to calculate the aerodynamic forces. They also divide the plane into several sections and each of this sections exerts forces and torques individually. Our implementation brings nothing different. The aircraft we simulate is Boeing-737-400.

This report is divided into three parts. Part one explains the physics model, part two explains the graphics, and part three consists of problems we encountered.

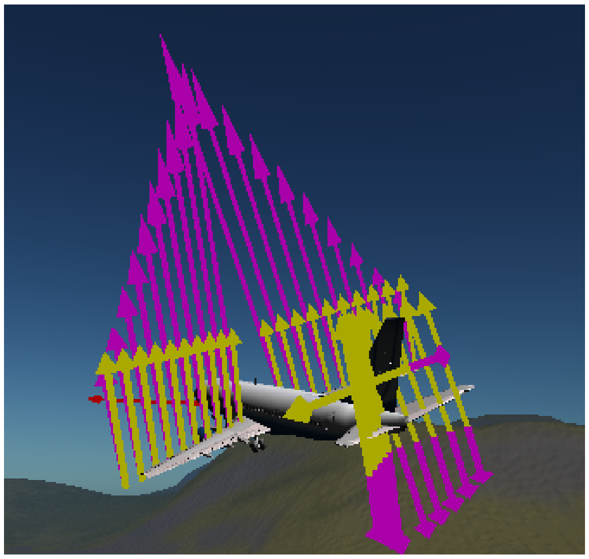

Fig. 2. The pink arrows represent aerodynamic forces

II. Physics model

Our Plane object consists of fuselage, main wings, elevators, rudder, engine, one front wheel, and two main wheels. It also contains information about plane's mass, inertia, position, velocity, angles, angular velocities, and junction points of wings and elevators. We can change the throttle of the engine, the control surfaces' deflection, and we can toggle brakes on and off. These fields and methods are considered enough to reasonably simulate a three-dimensional flight.

The simulation takes place in plane's update function which calculates the four forces, final torque and updates the state of the plane (position and orientation). These fields are then displayed as a model of the plane with pink arrows which represent the aerodynamic forces exerted on the plane (see Fig. 2). The yellow arrows are normals of the corresponding airfoil segments.

A. Forces

The forces are stored as three-dimensional vectors. The weight force is simulated simply as a constant force directed downwards and thus its vector representation is (0, -**weight**, 0), where **weight** is a multiplication of plane's mass and gravitational constant. The thrust force is obtained from Engine object which has predefined maximum thrust force and throttle level. The throttle level can be considered as the percentage of maximum thrust force that engine finally exerts.

The two remaining aerodynamic forces are not so easily calculated. The main wings and elevators are split into 10 and 5 sections respectively, so that each section produces its own lift and drag. This is important because the tip of an airfoil has different angle of attack compared to its root during rolling, which results in different lift forces in each section. The lift and drag from airfoils (including fuselage) are calculated using these equations:

$$p = 0.5 * \rho * v^2$$

$$F_{lift} = p * A * C_l * C_g$$

$$F_{drag} = p * A * C_d$$

where p is dynamic pressure, ρ is air density, v is the velocity of an airfoil segment, A is the area of an airfoil segment, C_g is the ground effect variable, and C_l and C_d are coefficients of lift and drag respectively. These coefficients depend of an angle of attack of an airfoil segment and are stored as tables for every airfoil segment. These tables were originally obtained from experiments in wind tunnels and are available via various books, particularly [1]. The ground effect variable ranges within [1, 1.15], reaching its max when the distance of airfoil from the earth is equal to zero, and reaches to its minimum after this distance passes 15 meters.

Our simulator also allows the plane to take off the ground and drive and land on it. For this, our Plane object has three Wheels objects, one front wheels object and two main wheels objects. The contact of wheels with the ground is determined by the plane's position, plane's orientation, junction point of wheels, and terrain height. We modeled wheels as springs.

The figure 3 shows a wheel and ground. When the wheel is not in contact with ground no resulting force is applied. When it is in contact, three forces are produced. For simplicity we assume that the force is always distributed equally among them. Plane is often pitched up such that its forward wheel is lifted, so it is not a very good assumption. The normal force always points up (even when plane is located on a slope) and has magnitude of y component of F(net force without wheels). When plane falls on ground normal force removes y component of acceleration, but doesn't remove y component of velocity so the plane "sinks" underground. To remove y component of velocity a spring force is calculated. Spring force's direction is obtained from plane's orientation and its magnitude is calculated by Hooke's law. Spring force by itself leads to oscillation of the plane once it lands. To counteract that, a damping force (not shown on the figure) is applied. Damping force is inversely proportional to the velocity of wheel. Finally, rolling friction force is calculated. It points in opposite direction of velocity. Its magnitude is $|N| * f_c$. Where N and f_c are normal force and a friction coefficient. The friction coefficient is different depending on the direction the wheel is moving in (figure 4). When planes hits the ground so fast that the wheels sink into the fuselage we consider that the plane has crashed.

After all individual forces are calculated we sum them into net force and compute the acceleration. Velocity is obtained by multiplying the acceleration by delta time, position is obtained similarly from velocity.

	Our simulation data	Real data
Take-off speed (m/s)	68	69
Take-off distance (m)	2100	2290

TABLE I
COMPARISON OF TAKE-OFF DATA BETWEEN OUR SIMULATION AND REAL WORLD

B. Torques

While forces decide the linear acceleration of the plane, torques affect its angular velocity. Airfoils are the source of the torque in the air. If the plane is on the ground, wheels also produce torque. A torque is calculated by computing the cross product of a force and the distance to the plane's center of mass. The angular acceleration is calculated using the following formula [18]:

$$aa = I^{-1} \cdot (T - av \times (I \cdot av))$$

Where aa, av, I, T are angular acceleration, angular velocity, inertia matrix, and torque respectively. Angular velocity and orientation are obtained in a similar way as the plane's velocity and position.

C. Simulation validation and controls

Our simulation allows controlling the plane by altering the flight control surfaces deflection level, throttle level, and braking. Control surfaces' deflection level changes their angle of attack which affects the amount of torque produced by them, thus controlling it allows controlling the orientation of the plane.

In order to make sure that our simulation is realistic, we compare certain aspects of the behaviour of our plane with the real plane's behaviour. In particular, we compared the take-off speed, take-off distance, behaviour of the plane in air with and without interference from our side, ability to land and stalling. The comparison of our take-off data and real data is shown in Table 1.

It can be seen that our simulated plane produces just the right amount of lift, drag and wheels friction forces. They allow to take-off only after gaining realistic speed and completing distance of a realistic runway. It was possible to perform basic maneuvers using controllable surfaces. We were able to land in the simulation, and descending at too high speed crashes the wheels, just like in real world. Flying straight up leads to stall and temporary loss of control, however plane eventually stabilizes itself facing downwards if the height is sufficient. This shows that the model of the plane, i.e. how airfoils are located with respect to each other is done correctly, as well as their lift and drag calculations. Flying without pilot's interference makes plane to enter a phugoid motion, thus the midair lift and torque are implemented correctly.

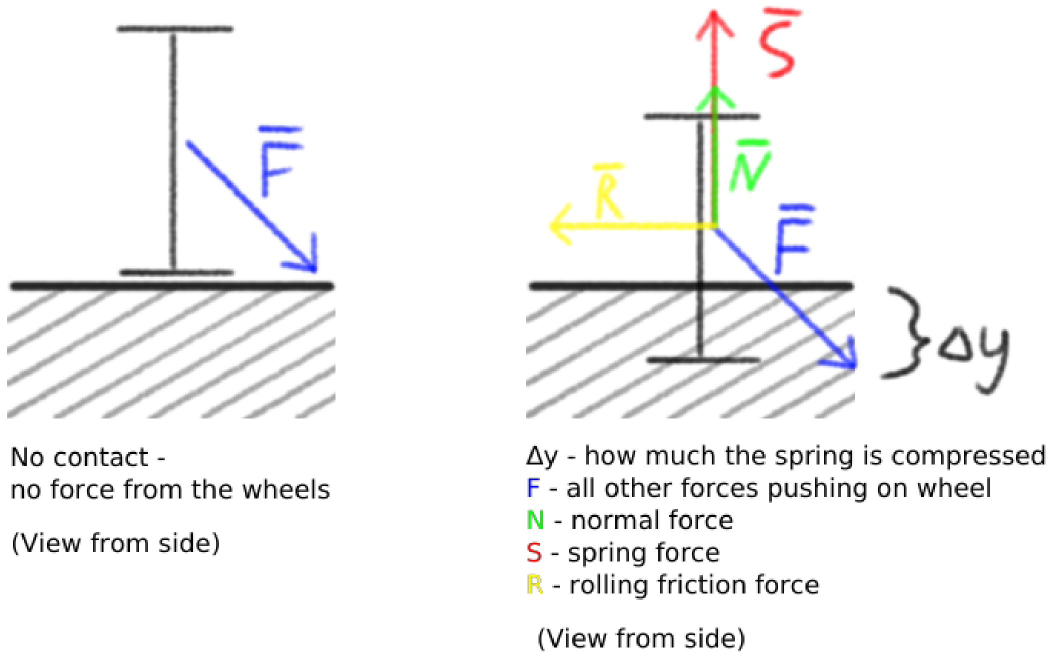

Fig. 3. Wheels modeled as springs

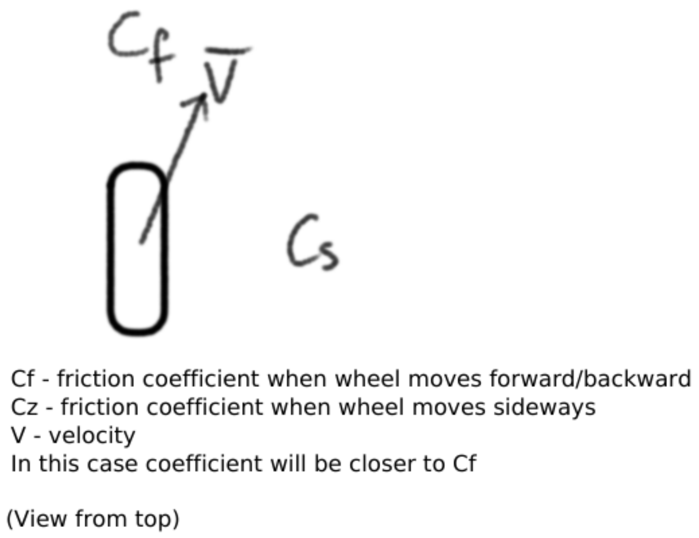
Cf
$\bar{V}$
Cs
Cf - friction coefficient when wheel moves forward/backward
Cz - friction coefficient when wheel moves sideways
V - velocity
In this case coefficient will be closer to Cf
(View from top)

Fig. 4. Wheels' rolling friction coefficients

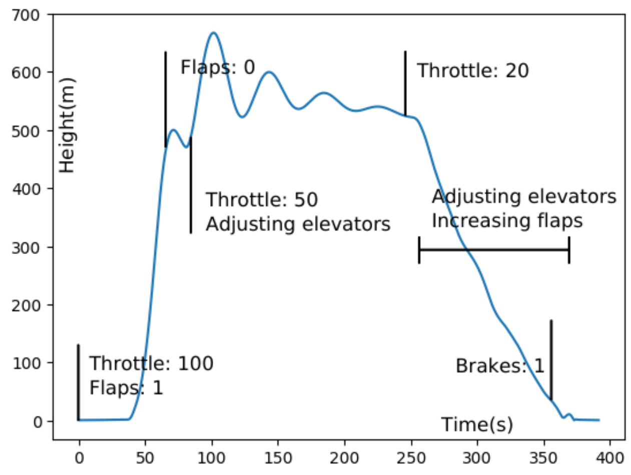

Fig. 5. The complete flight of our plane from takeoff to landing.

III. Graphics

The graphics were done using SFML [8], OpenGL [12], and GLM [5] libraries. SFML is used to open a window and poll input events. OpenGL provides access to hardware acceleration and graphics pipeline. Graphics pipeline takes data(vertices, faces, texture coordinates) passed from CPU, shaders, and runs the shaders on data. It automatically discards vertices that are out of the field of view in a process called clipping. GLM provides implementations of common linear algebra constructs and operations as well as utility functions for using them with OpenGL.

GLSL shaders don't have functionality to import/include code from other shaders. Because of that code for calculating lighting was duplicated among shaders for models/terrain. One possible solution is to implement some kind of preprocessor for GLSL. The other solution is deferred rendering. The idea is to do rendering in multiple stages. In the first stage models/terrain/other do some work specific to them and output data needed to compute lighting into textures (figure 6). In the second stage, a lighting shaders takes those texture and outputs the final image to the screen.

The lighting shader uses a simple Phong lighting model [3] popular in older video games. It splits the effect of light into 3 parts: ambient lighting, diffuse lighting, and specular lighting. Ambient lighting approximates indirect light (when the light rays don't come from the light source directly but are reflected by surrounding objects) by making models slightly lit up by default. Diffuse lighting simulates direct light by making a part of the surface of the object slightly darker or lighter depending on the amount of light that falls on it (a light ray is most intense when it hits at 90° and least intense when it hits at 0° / 180°). Specular lighting approximates the reflection of direct light. Besides the angle of light rays and the angle of the camera specular lighting also depends on shininess of the surface. Shininess is just a scalar.

After the ambient, diffuse, and specular lighting are calculated they are summed and multiplied by the color of the surface. Finally, the color is blended with the atmosphere color based on distance from the camera and with cloud color if the object is occluded by a cloud.

A. *Atmospheric scattering*

We adapted open source implementation of atmospheric scattering from [19]. In flightsim the Earth is flat. But when we render the sky we assume that we are surrounded by a sphere that designates the "edge" of the atmosphere. For every pixel we "shoot" a ray from the camera to sky. Then we "march" along that ray with a small step size and compute how much light reaches the camera from every point on the ray. We do this for 3 coefficients and 2 phase functions that depend on wavelength (red, green, blue) and type of air particles (air molecules or larger particles like dust). Finally, we sum them together to obtain the final color.

The amount of light that reaches the camera from a point in the sky is a product of 2 quantities.

1) The amount of light at the point. It is the amount of light that reaches the point from the sun. To compute it we need to shoot another ray from this point to the sun. This quantity is given by

$$T(X, P_a) = exp(-\beta \sum_{X}^{P_a} exp(-h/H)ds)$$

Here, X is the point in the sky. P_a is the position where the ray from the point to sky intersects with the edge of the atmosphere. β is a coefficient that depends on wavelength and air particle type. h/H is a scaled height (0 at ground, 1 at the edge of the atmosphere). ds is a small distance along the ray from the point to the sun.

2) The ratio of light at the point that reaches the camera. It is given by

$$T(P_c, X) = exp(-\beta \sum_{P_c}^{X} exp(-h/H)ds)$$

Here, P_c is the position of the camera. ds is a small distance along the ray from the camera to the point.

The final equation to compute the color of the sky at a given pixel is

$$skycolor(P_c, P_a) = sunintensity \cdot P(V, L) \sum_{P_c}^{P_a} T(P_c, X)T(X, P_a)ds$$

Here, P_a the position where the ray from the camera to sky intersects with the edge of the atmosphere. ds is a small distance along the ray from the camera to the sky. V is the direction of the ray from camera. L is the direction of sunlight. The phase function $P(V, L)$ specifies how much light gets scattered depending on the angle between V and L. $sunintensity$ is just a constant.

Here are phase functions for air molecules and larger air particles.

$$P_R(V, L) = \frac{3}{16\pi}(1 + (V \cdot L)^2)$$

$$P_M(V, L) = \frac{3}{8\pi} \frac{(1 - 0.76^2)(1 + (V \cdot L)^2)}{(2 + 0.76^2)(1 + 0.76^2 - 2 \cdot 0.76 \cdot V \cdot L)^{\frac{3}{2}}}$$

B. *Clouds*

The rendering of clouds consists of 2 stages.

- Rendering their models as if they were opaque models and computing information needed for the second stage.
- Blending them into the final image.

To render the clouds we generated 2 identical meshes shaped like discs. They consist of 100 concentric rings each with 100 segments and cover a wide area of sky around the plane. The vertices of the first mesh are offsetted according to heightmap of the top of the cloudscape. The vertices of the seconds mesh are offsetted according to heightmap of the bottom of the cloudscape. These 2 meshes are then rendered twice. The first time they are rendered with ordinary depth order (fragments that are closer are drawn on top of fragments that are farther). The second time they are rendered with the opposite rendering order. This gives us the farthest position of a cloud and the closest position of a cloud for any given pixel(figure 8) when the camera is outside of the clouds. When camera is inside the clouds the renders with different depth orders would coincide. We handle this case by simply detecting when the camera is inside a cloud and setting the nearest position to the camera's position. To blend the the clouds into the final image we compute how transparent they should look. It is done by subtracting the farthest position of a cloud from the closest and plugging them into Lambert-Beer law.

$$finalcolor = lerp(finalcolor, exp(-\sigma(farpos - nearpos))$$

Here is σ an extinction constant that determines how dense the clouds are.

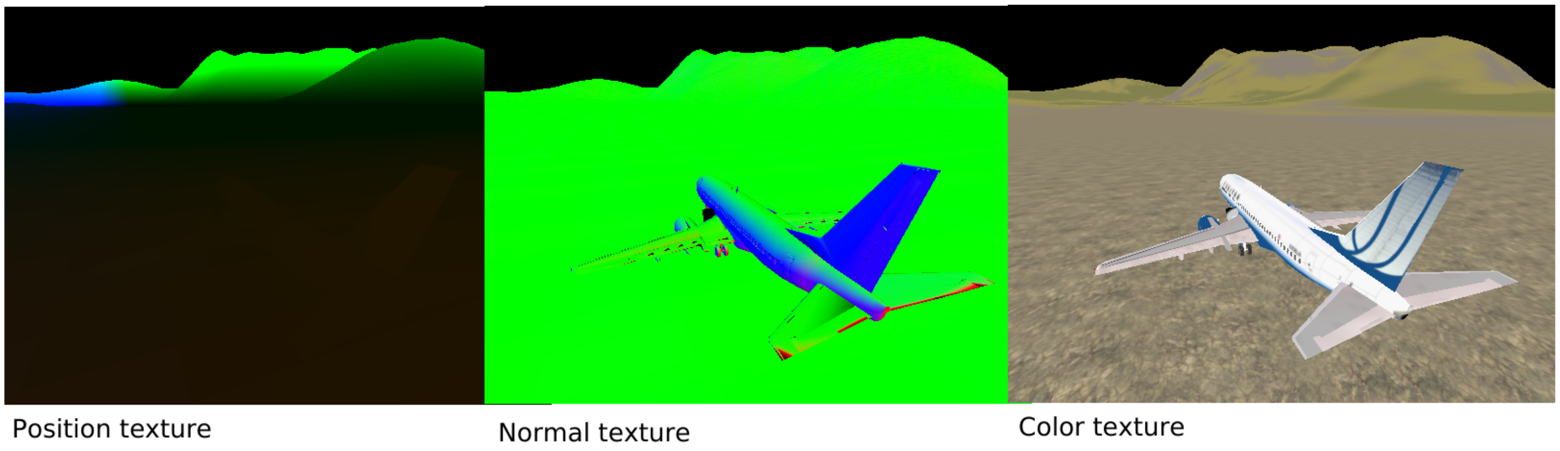

Fig. 6. Textures computed in the first stage of deferred rendering

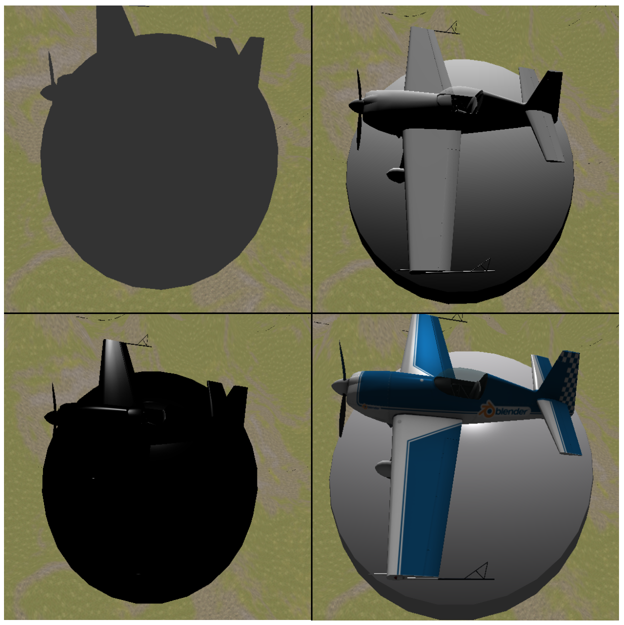

Fig. 7. Phong lighting model (ambient, diffuse, specular, sum with color). Note: this not the airplane we are modeling. It's for testing graphics.

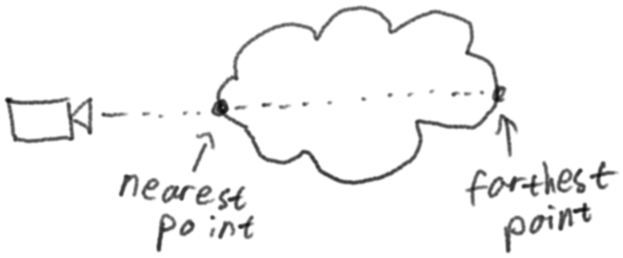

Fig. 8. Rendering a cloud

C. Wavefront file formats

We chose Wavefront OBJ [15] and Wavefront MTL [16] file formats for storing models. They are simple text files and were designed to be easy to parse. We wrote simple parsers for them. The parsers ignore information needed for the features we haven't implemented. Obj files store geometry (vertices, faces) and texture coordinates. They also specify which material to use for each face. Mtl files store the materials. Materials specify texture file paths, default colors to use in the absence of textures, shininess for specular lighting, and which interpretation of the Phong lighting model to use. For now, we only have a single version of Phong and it doesn't completely correspond to any of the models of Wavefront MTL. But all of the Wavefront MTL lighting models provide all the information needed for our model so we just load what we need and ignore the rest.

D. AC3D file format

In addition to wavefront OBJ file format flightsim supports AC3D models. There reason for choosing this format is that AC3D is used in FlightGear; Therefore, open source plane models can be loaded into flight simulator. Another advantage of this format is that it is simple text file as OBJ. While writing a parser the design of the model class was changed. Currently our models consist of separate arrays for materials, textures and objects. Object is a structure that contains information about mesh like vertices, material indexes, texture indexes. Objects have a hierarchy that allows to correctly apply transformations. For example, we can rotate root object and all child objects will be affected. In order to decrease the amount of memory taken by models on a gpu, only unique vertices are kept in memory and drawing is performed using list of their indexes kept on element buffer.

E. 3D grid

For testing purposes while implementing physics of flight in 3D, terrain was replaced by a flat surface with a grid. The main reason for implementing it was that plane flight could have been tested without the fully implemented terrain. At that time we didn't have collision with terrain and the terrain wasn't big enough. The grid is of a fixed size and always in the same position relative to the plane. By using shader described in (http://madebyevan.com/shaders/grid/) illusion of the movement is created.

F. Arrows

For debugging the physics of the flight, the way of representation of vector quantities like velocity and force was required. It was done by simple arrows. The application holds only one instance of arrow in memory which is transformed to different positions and orientations.

G. Terrain

The geometry of terrain is stored in a grayscale image called heightmap. Each pixel in a heightmap represents height data for $32m^2$, so a map with a resolution of 2048x2048 corresponds to terrain with area 65536^2m^2. Our program uses this heightmap data to generate polygons. In order to reduce the complexity of rendering we use tessellation shaders to achieve different levels of detail based on the distance to the camera.

Tessellation shaders are part of rendering pipeline introduced in OpenGL from version 4.0 [13]. These shaders accept a sequence of vertices called patches that represent some primitive geometry(squares in our case) and then subdivide them into more primitives. The pipeline consists of three parts. Tessellation Control Shader (TCS) that specifies the way patch will be divided by choosing *tessellation level* for edges of a patch. Tessellation Primitive Generator generates primitives and is not programmable by a user. Tessellation Evaluation Shader (TES) processes all vertices produced by generator - applies MVP(Model view projection matrix) and sets correct height taken from heightmap texture.

Regions of terrain closer to the camera are divided into smaller patches. Initially, we increased tessellation levels for closer (and smaller patches), but we found that level of detail is sufficient if all patches have roughly the same level of tessellation.

Rendering of terrain starts by dividing it into non-uniform squares. This is achieved by representing it as a quadtree structure. We start from the root which is a whole terrain and then split it into four child nodes. Each child node is recursively split until nodes get small enough relative to the camera position. This is done by comparing the distance to the camera with diagonal of a node. This process is repeated every frame.

Then for each leaf node of the quadtree the renderer calls a draw operation and sends a square patch. Vertex shader simply passes them to TCS. TCS sets the tessellation levels for each edge of a patch. TCS passes a square patch and tessellation levels of each edge to TPG. TPG tessellates the patch into polygons and passes them to TES.

Next, TES shader receives a set of polygons that form a square. In this stage, shader accesses heightmap texture

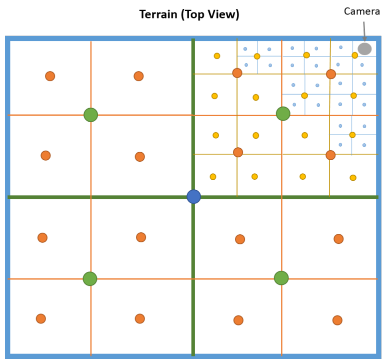

Fig. 9. Quadtree representation of the terrain. [14]

and based on its intensity it calculates height for vertices of polygons. Then it applies MVP matrix to transform vertices to the screen coordinates. In addition to the vertices in screen coordinates, TES outputs 2 vectors: texture coordinates (also referred to as UV coordinates) and the same vertices in world coordinate space.

Now that the geometry of terrain was calculated and projected to screen coordinates it has to be rasterized. The terrain is textured differently than models. A model can be wrapped completely with few textures. The terrain is too large so instead, it is tiled by 4 repeating small textures [10]. Multiple textures can be blended together. In order to prevent visible tiling on terrain caused by texture repetitions, a technique described in [9] is used. An image called alphamap determines the "presence" of each texture on each spot of the terrain. Red, green, blue, and alpha channels correspond to 4 textures. Texture coordinates from TES are used to sample 4 colors from 4 textures. World coordinates from TES are used to sample alphamap to get 4 coefficients for interpolating colors.

$$\overrightarrow{c_1} = sample(texture_1, t_x, t_y)$$
$$\overrightarrow{c_2} = sample(texture_2, t_x, t_y)$$
$$\overrightarrow{c_3} = sample(texture_3, t_x, t_y)$$
$$\overrightarrow{c_4} = sample(texture_4, t_x, t_y)$$
$$r, g, b, a = sample(alphamap, w_x, w_y)$$
$$\overrightarrow{color} = r \cdot \overrightarrow{c_1} + g \cdot \overrightarrow{c_2} + b \cdot \overrightarrow{c_3} + a \cdot \overrightarrow{c_4}$$

In the current version of the program, we use a heightmap, alphamap, and textures that are generated using L3DT [2].

IV. Problems

A. Clouds

There are 2 problems with clouds.

- When multiple clouds are lined up on your line of sight their transparency is wrong.
- Clouds look wrong when the sun is down.

The first problem is because we just subtract the nearest cloud point from the farthest cloud point to compute the

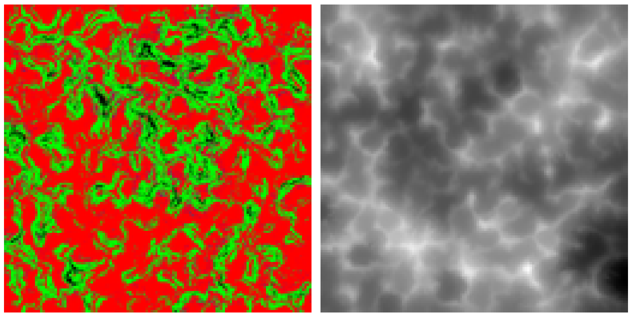
Fig. 10. Alpha map (with alpha layer removed to visualize) and heightmap.

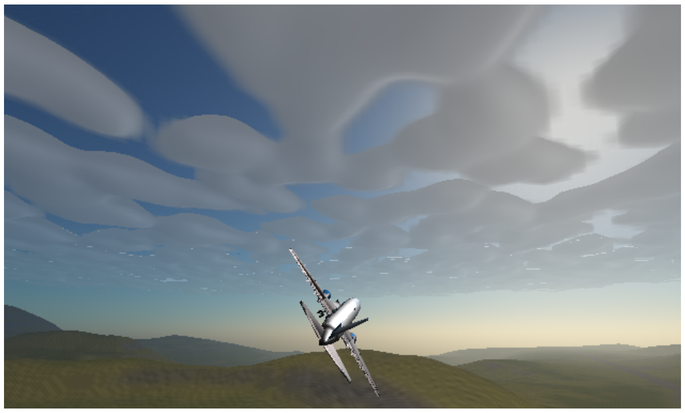
Fig. 11. Clouds

”thickness” of the cloud. It doesn’t take into account that there might be gaps of air between these points (when the nearest and farthest points belong to different blobs of clouds).

The second problem is that if we render our clouds as described in the explanation of clouds the parts in the shadow would look completely dark. The reason is that we are not simulating the ”in-scattering” effect. It is when a light ray enters the cloud and bounces around inside the cloud making it glow. It is very hard to simulate efficiently, so instead we just faked the glow by making the cloud brighter. Because of that, the clouds look bright even when the sun is down.

B. Normals of AC3D models

One of the problem encountered during loading plane models in ac3d format was the fact that they don’t contain precalculated normal vectors. Calculations had to be done during loading and the problem was caused when one vertex was shared by multiple faces. Averaging normals of all adjacent faces caused artifacts where after light calculations vertex became dark. Partially the problem was resolved by taking into account the area of each face during normal computation (figure 12).

C. Interface

The use of sfml functions simultaneously with the openGL context of the application caused problems. The main one was the hardware support of correct version of opengl on the computers of our team members. SFML relies on openGL calls that are deprecated and hardware of one team member didn’t support it. Moreover, before calling its functions state of opengl had to be saved in stack and then switched back after interface was drawn. It frequently was the cause of graphical artifacts after major changes in the rendering system.

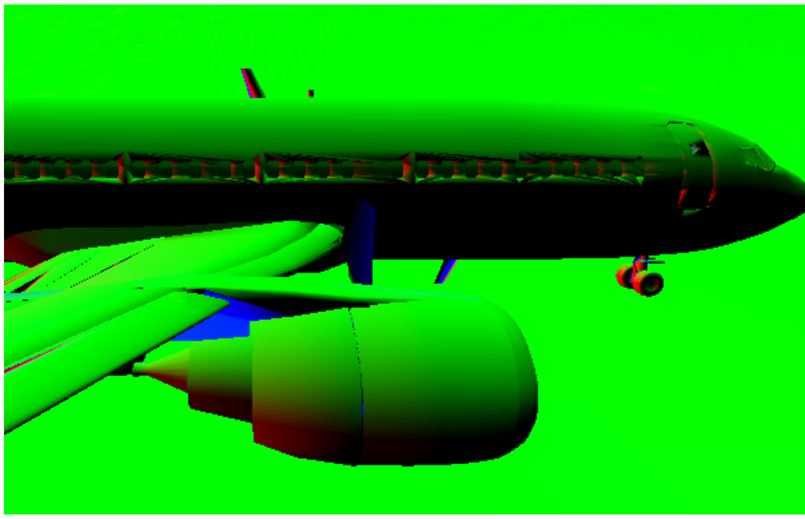
Fig. 12. The averaged normals of an AC3D model.

D. Shadows

We didn’t implement shadows. To generate shadows we need to render the scene twice. Once from the point of view of the camera and once from point of view of the sun to find which objects get hit by light rays. For the second part we only need to render the depth information. But doing so would require us to write a second set of shaders which we didn’t have time for. Reusing existing shaders would be too computationally expensive because normals, colors, etc. would be computed twice.

E. Debugging in OpenGL

Code that runs on CPU can be debugged with a debugger or print statements. Shaders can’t even contain print statements. The only way to debug them is to display the needed info visually, e.g map numbers to colors.

F. Cracks in terrain

Our terrain consists of nonuniform patches and in the edges, where patches of different sizes meet, visual artifacts occur. One solution suggested in a paper from Nvidia [6] is to reduce tessellation level by two times in the smaller patch’s edge that is adjacent to the larger patch. However, in our implementation, there were still some visible cracks in the terrain that might have been caused by incorrect tessellation level calculation. We improved the quadtree neighbor finding algorithm, but that didn’t solve the problem completely.

References

[1] A.C.Kermode, ”Mechanics of Flight”.

[2] A. Torpy, ”About L3DT”, Available: http://www.bundysoft.com/docs/doku.php?id=l3dt:about.

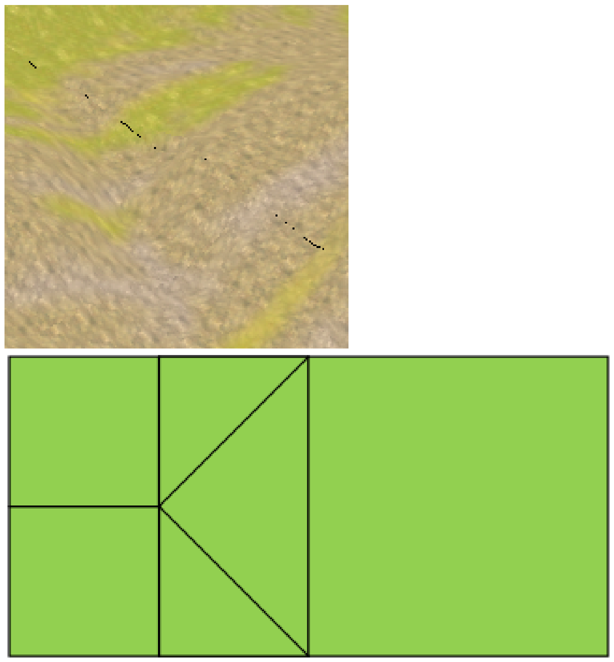
Fig. 13. Crack in terrain; Solution with tessellation level reduction.

[3] Bui Tuong Phong. 1975. ”Illumination for computer generated pictures”.

[4] FlightGear Developers & Contributors, ”Flightgear Flight Simulator sophisticated, professional, open-source”, Available: http://home.flightgear.org/.

[5] G-Truc Creation, ”OpenGL Mathematics”, Available: https://glm.g-truc.net/0.9.9/index.html.

[6] I. Cantlay, ”Directx 11 terrain tessellation”, *Nvidia whitepaper*, January 2011.[Online] Available: https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/TerrainTessellation_WhitePaper.pdf.

[7] Laminar Research, ”How X-Plane Works”, Available: https://www.x-plane.com/desktop/how-x-plane-works/.

[8] L. Gomila, ”Simple and Fast Multimedia Library”, Available: https://www.sfml-dev.org/.

[9] Q. Inigo, ”Texture Repetition”. [Online]. Available: http://www.iquilezles.org/www/articles/texturerepetition/texturerepetition.htm. [Accessed: 19- Nov- 2018].

[10] RasterTek, ”Terrain Texture Layers”, Available: http://www.rastertek.com/terdx10tut17.html.

[11] The Khronos Group, ”Cubemap Texture”, Available: https://www.khronos.org/opengl/wiki/Cubemap_Texture.

[12] The Khronos Group, ”OpenGL: The Industry’s Foundation for High Performance Graphics”, Available: https://www.opengl.org/.

[13] The Khronos Group, ”Tessellation”, Available: https://www.khronos.org/opengl/wiki/Tessellation.

[14] V. Bushong, ”Tessellated Terrain Rendering with Dynamic LOD”, Available: https://victorbush.com/2015/01/tessellated-terrain/.

[15] Wavefront OBJ specification, Available: http://www.martinreddy.net/gfx/3d/OBJ.spec.

[16] Wavefront MTL specification, Available: http://paulbourke.net/dataformats/mtl/.

[17] Wikipedia, ”Aircraft principal axes”, Available: https://en.wikipedia.org/wiki/Aircraft_principal_axes.

[18] David Bourg, ”Physics for Game Developers”

[19] Scratchapixel, ”Simulating the Colors of the sky” Available: https://www.scratchapixel.com/code.php?id=52&origin=/lessons/procedural-generation-virtual-worlds/simulating-sky