Nicolas Hsu | nh6534

Physical Simulation: Final Project

For my project, I attempted to implement a 4D rigid body simulation using plane-based geometric algebra ,as described by Dorst & Keninck (https://bivector.net/PGAdyn.pdf).

To render the 4D scene, I implemented a ray-marcher that can be toggled between 3D and 4D. (Disclaimer: I implemented the 3D version of this ray marcher as my final project for Dr. Vouga's graphics course). Visualizing 4D is typically done using various methods of projecting from 4D to 3D to the 2D screen, most of which should be equivalent to ray marching. I'll focus the rest of this report on the physics.

Plane-based geometric algebra (PGA) can be thought of as a framework that attempts to unify a bunch of different geometric operations under a few highly structured operations. These operations include translations, rotations, reflections, projections, meet/join, orthogonal complements, and more. PGA extends naturally to higher dimensions, which makes it a suitable candidate for this project, especially because there isn't much popular code that handles math in dimensions higher than 3. PGA is not a fundamentally new idea; it combines ideas from screw theory, Lie groups, dual quaternions, and others.

The fundamental unit in N-dimensional PGA is a plane. The basis elements e0, e1, … eN are defined such that e0^2=0 and ei^2=1 for all i>0. These elements also anticommute: ei * ej = -ej * ei. There are various other products such as the outer product, inner product, regressive product, etc. which are defined in a specific way so that they can be used to represent the aforementioned operations. Intuitively, this is possible because all rigid motions can be represented as a sequence of plane reflections, and all fundamental geometric objects (points, lines, planes) can be represented as the invariant spaces of these transformations.

By making judicious use of PGA operators, we can obtain equations of motion which unify translation and rotation, which is convenient for extending into higher dimensions. In N dimensions, the number of degrees of rotational freedom is N choose 2, which quickly becomes difficult to implement in code. PGA simplifies this process.

Newton/Euler	$\dot{P}$	$=$	F
Inertia	$\mathsf{I}_b[B]$	$\equiv$	$\Sigma_i\, m_i\, X_i \vee (X_i \times B)$
Dynamics	$\dot{B}_b$	$=$	$\mathsf{I}_b^{-1}[B_b \times \mathsf{I}_b[B_b] + F_b]$
Kinematics	$\dot{M}$	$=$	$-\frac{1}{2}\, M\, B_b$

Table 2.1: *The equations of motion in PGA.*

Nicolas Hsu | nh6534

Implementation overview

The main library components are Rust (https://www.rust-lang.org/), WGPU (https://wgpu.rs/) and WGSL (https://www.w3.org/TR/WGSL/), and the geometric-algebra crate (https://crates.io/crates/geometric_algebra).

Rust is a programming language for the CPU-side code. WGPU is a graphics framework with many similarities to Vulkan, but at a higher level of abstraction. It can run on top of Vulkan, Metal, WebGL, OpenGL, and many other backends. WGSL is a shading language, with similar purpose as GLSL.

Geometric-algebra is included as the math library generator. For generating WGSL I've locally merged Github user AndrewBrownK's pull request (https://github.com/Lichtso/geometric_algebra/pull/5) into Github user Lichtso's main branch.

The dimension can be toggled in Cargo.toml via the "four_d" feature. See https://doc.rust-lang.org/cargo/reference/features.html for more information.

The rest of the program is fairly standard. Set up a render pipeline, maintain the state of the camera/objects/lights in the scene, pass them into the shader via a uniform buffer. Movement and camera controls are handled in the main loop. The main program is in main.rs, and the shader code is in shader.wgsl.

some references: https://bivector.net/PGAdyn.pdf, https://bivector.net/doc.html, https://bivector.net/PGA4CS.pdf, https://arxiv.org/pdf/2002.04509.