Lucid3D Revised Overview

# Abstract

Lucid3D is being designed from the ground up to allow game developers to achieve their game visions. If you can dream it, Lucid can help you build it. To enable this, Lucid provides three distinct pieces: 1) A flexible and customizable core engine, 2) a powerful tool chain for authoring content, assets, and game logic, and 3) a core library of functionality supplied with the engine to get game developers started.

# Primary Customers

Lucid3D is designed with 2 primary customer groups in mind:

1. High level game developers. High level game developers have a vision for a game, and have artists to help build the assets. They want to write *game logic*, and define what they believe to be a fun game. Lucid delivers a full end to end tool-driven experience for these developers, and they shouldn’t have to touch any engine level source code to realize their goals.
2. Mid to low level game developers. Because Lucid’s core is exposed at certain extension points, a native developer who wishes to replace components (such as physics engine, renderer, etc…), has the hooks to do so. This allows a larger team which wants more rich functionality to achieve their goals as well.

# High Level Goals and Feature Requirements

The highest priority goals are:

1. Provide a solid, core engine architecture, with each replaceable component behind a very strict and well documented contract. This means that any customer who wishes to entirely replace a subsystem can, as long as their component matches the contract and requirements.
2. Provide a tooling abstraction above the engine, so that there is clear interface between tools, design-time interaction, and run-time components. In effect, the customer could also write some of their own custom tools to this interface, so long as they upheld formats and other constraints, which will be well defined and documented.
3. With the core components and clear tool division, an entire package will be delivered in the default Lucid3D SDK. Licensing options may restrict how much access to the underpinnings the developer gets. For instance, the indie license may only give surface level access via the tool, and not release the headers and libs for building components.
4. An empty architecture is only somewhat useful. Lucid will ship with some of the engine components implemented (graphics, input, audio) and will likely package in 3rd party components for some of the other components (physics, UI, scripting). Which components fall into which bucket is still to be determined. The very core of the engine (the glue in between) is what Lucid’s primary product is, and is the only piece not replaceable. Users can override the renderer, the input stack, etc…

# Feature Areas

The core feature areas for the first release of Lucid3D are:

1. Defining the overall architecture. How do components fit together? How and where do we define the lines so that entire subsystems can be replaced? How do we keep things decoupled enough so that we can pull out the entire graphics stack without breaking the rest of the engine? How do we define the tooling layer so that it’s not dependent on details of the engine?
2. Defining the core formats. Meshes, animations, fonts, sound effects. All of these types of assets, and many more, will be consumed by the runtime, and authored in the editor. We need to define these well, and provide details and great implementations out of the box. We may even want to build intermediate libraries for persisting and reading the formats, so that tool and engine component developers can leverage them more easily.
3. Building the first round of components. We’ve decided that the first release of Lucid will include some of the core components, though there is not enough time and bandwidth to deliver them all. We’ve chosen graphics, input, and audio to be included in our first release. Other 3rd party components, such as physics, will either be prepackaged (if their licenses allow), or offered as suggestions for customers to use.
4. Building the tool chain. A strong, easy to use toolset is what will allow Lucid to compete with our competitors, such as Unity. All of our competitors have very streamlined all-in-one WYSIWYG editors, which allows non-programmer game designers to build compelling games. This is likely our biggest challenge and largest amount of work for the first release. In many ways, the editor must be far ahead of our engine, leading the way in defining formats and user scenarios. The editor will drive things like packaging, optimizing of data, defining streaming regions, baking lighting, building LOD for objects and environments, and more. The runtime is really just that, a system which executes all the great data coming out of the tools.

# Core Design Principles

The primary principle, across the board for the project, is to only deliver exactly what the customer needs to perform their work. The less we expose, the more flexible the components can be in their implementations. The more flexible they are, the easier it is for customers to replace components with their own. The clearer the lines between pluggable pieces, the better off we’ll be in the long run.

Component boundaries should be fairly coarse, which will help keep the boundaries better defined. For instance, if someone is going to replace the graphics stack, they must replace the entire stack. The material and shader system is not a separate component from the renderer. Trying to separate things too granularly leads to awkward interfaces and unnecessary layering between what should be very tightly integrated pieces. This also affords us a lot of flexibility with things like internal formats for data, optimizations across subcomponents, and many other benefits.

This also leads to a fewer number of interfaces that we expose to the customer, again circling back to our first principle. Ideally, the customer would only see a single top level interface for each subsystem, which is its point of extensibility (where replacement can occur). Each component talks to each other only through this interface, which guarantees loose enough coupling that the components can be individually replaced. This also allows entire components to be stubbed during development, easing the process of getting the basics up and running.

The tools layer will also only talk to the engine through a set of well-defined points. If needed, there will be a separate library of helper utilities which are shared between the engine and the tool, and which should be made available to the customer which wishes to write their own component. The types of helpers which belong here are file format reading and writing, compression and decompression methods, and other functionality which has a proper place in both the tool chain and runtime, and which game developers would need access to in order to write a competent replacement to a component.