Living Architecture Systems Description

Extensible Spatial Data System For Responsive Architectural Environments

Kevan Cress and Timothy Boll
Living Architecture Systems Group

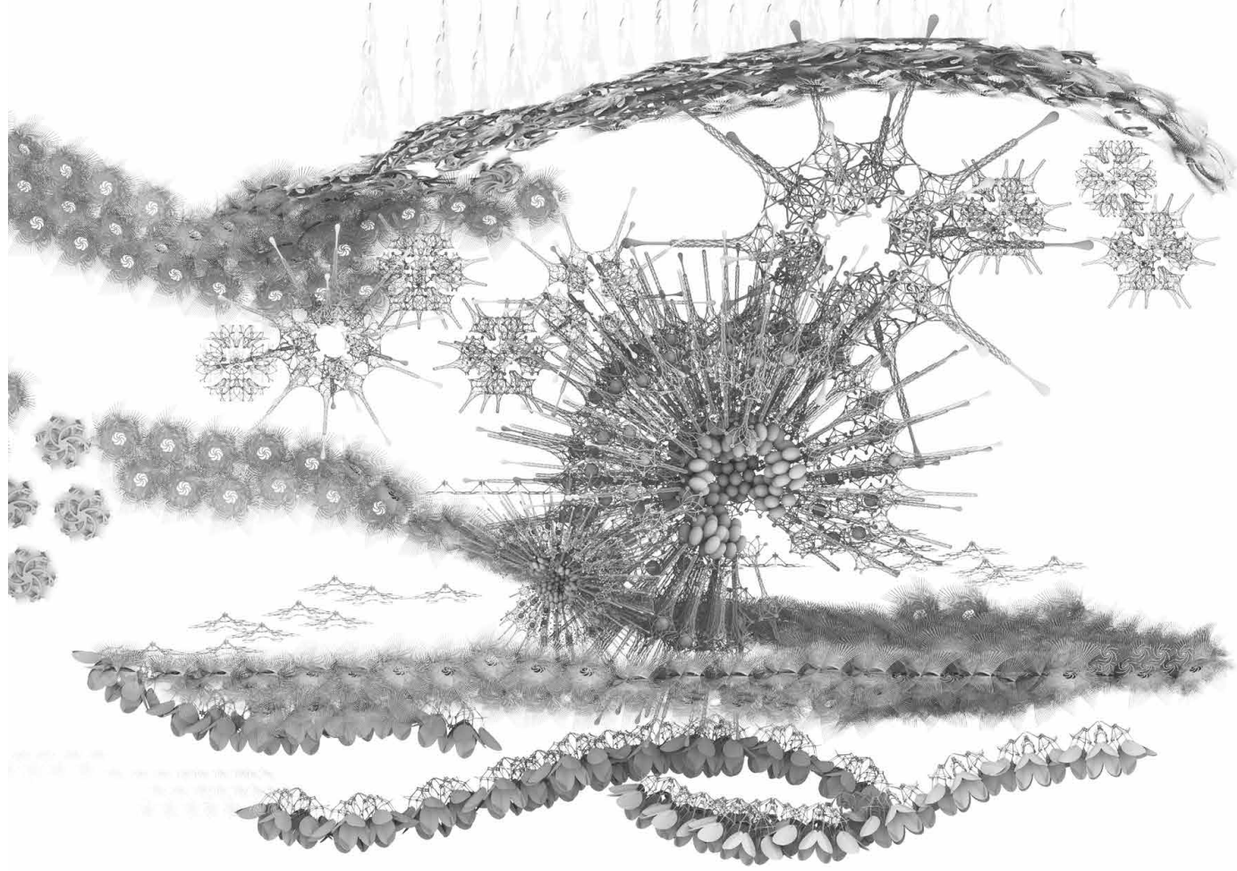
The Living Architecture Systems Description is a data framework that describes the organization of components and assemblies within a series of experimental responsive architectural environments. These specialized environments have been developed by the Living Architecture Systems Group, an international partnership of researchers, artists, and industrial collaborators studying how we can build living architectural systems. The Living Architecture Systems Description is designed to support computational modeling and time-based media development and control for these environments. The publication provides core descriptions including background, definitions, and case studies, and offers information on coordinating the data framework with common digital content creation tools in order to facilitate data exchange between specialized software developed for living architecture projects and common software modeling platforms The Systems Description provides a framework that can be used to describe a testbed's hierarchies and organization by labeling and editing living architecture systems data within 3D model environments. This provides a common language that supports the transfer of digital models between a variety of digital content creation tools.

The Systems Description enables designers to explore the emergent properties of layered components and systems as they create new interactive and responsive environments. The data framework offers a fundamental 'syntax' that can support conceptions of living architecture, equipping a new generation of designers, thinkers, and builders with skills needed to work within complex, turbulent environments.

ISBN 978-1-988366-44-9

9 781988 366449 >

Riverside Architectural Press



Living Architecture Systems Description

EXTENSIBLE SPATIAL DATA SYSTEM
FOR RESPONSIVE ARCHITECTURAL ENVIRONMENTS

KEVAN CRESS AND TIMOTHY BOLL
LIVING ARCHITECTURE SYSTEMS GROUP

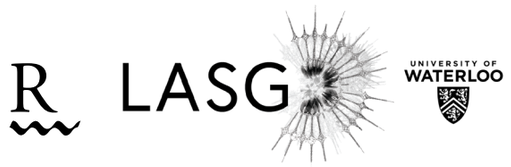
LASG

UNIVERSITY OF WATERLOO

Living Architecture Systems Description

EXTENSIBLE SPATIAL DATA SYSTEM
FOR RESPONSIVE ARCHITECTURAL ENVIRONMENTS

KEVAN CRESS AND TIMOTHY BOLL
LIVING ARCHITECTURE SYSTEMS GROUP



Published by Riverside Architectural Press
www.riversidearchitecturalpress.ca

Published in February 2023

Title: Living Architecture Systems Description: Extensible Spatial Data System for Responsive Architectural Environments
Names: Cress, Kevan, 1995-author. | Boll, Timothy, 1991-author. | Gorbet, Matt, 1973-contributor. | Lancaster, Michael, 1995-contributor. | Beesley, Philip, 1956-editor. | Living Architecture Systems Group, issuing body.

Description: This document outlines the design and use of a spatial data framework for the interchange of systems-design data relating to testbed environments, software, and interfaces designed and built by the Living Architecture Systems Group.

ISBN 978-1-988366-44-9

Design and Production by Living Architecture Systems Group

Social Sciences and Humanities Research Council of Canada
Conseil de recherches en sciences humaines du Canada
Canada

About the Living Architecture Systems Group

This publication forms part of a series of work-in-progress reports and publications by Living Architecture researchers and contributors. The Living Architecture Systems Group is an international partnership of researchers, artists, and industrial collaborators studying how we can build living architectural systems–sustainable, adaptive environments that can move, respond, and learn, and that are inclusive and empathic toward their inhabitants. "Smart" responsive architecture is rapidly transforming our built environments, but it is fraught with problems including sustainability, data privacy, and privatized infrastructure. These concerns need conceptual and technical analysis so that designers, urban developers and architects can work positively within this deeply influential new field.[1] The Living Architecture Systems Group is developing tools and conceptual frameworks for examining materials, forms, and topologies, seeking sustainable, flexible, and durable working models of living architecture.

Living Architecture Systems Group research is anchored by a series of prototype testbeds: accessible, immersive architectural sites containing experiments and proof-of-concept models that support living architecture as a practical model for our future built environment. These testbeds can help researchers answer ethical, philosophical and practical questions about what living architecture means and who it is for within our societies and environments, creating sites of collaborative exchange that act both as research ventures and as public cultural expressions.

A series of far-reaching critical questions can be explored by using the tools and frameworks that are described within this specialized publication series: can the buildings that we live in come alive? Could living buildings create a sustainable future with adaptive structures while empathizing and inspiring us? These questions can help redefine architecture with new, lightweight physical structures, embedded sentient and responsive systems, and mutual relationships for occupant that provide tools and frameworks to support the emerging field of living architecture. The objective of this integrated work envisions embodied environments that can provide tangible examples in order to shift architecture away from static and inflexible forms towards spaces that can move, respond, learn, and exchange, becoming adaptive and empathic toward their inhabitants.

1 Kas Oosterhuis and Xin Xia, iA #1, Interactive Architecture (Rotterdam: Episode Publishers, 2007)

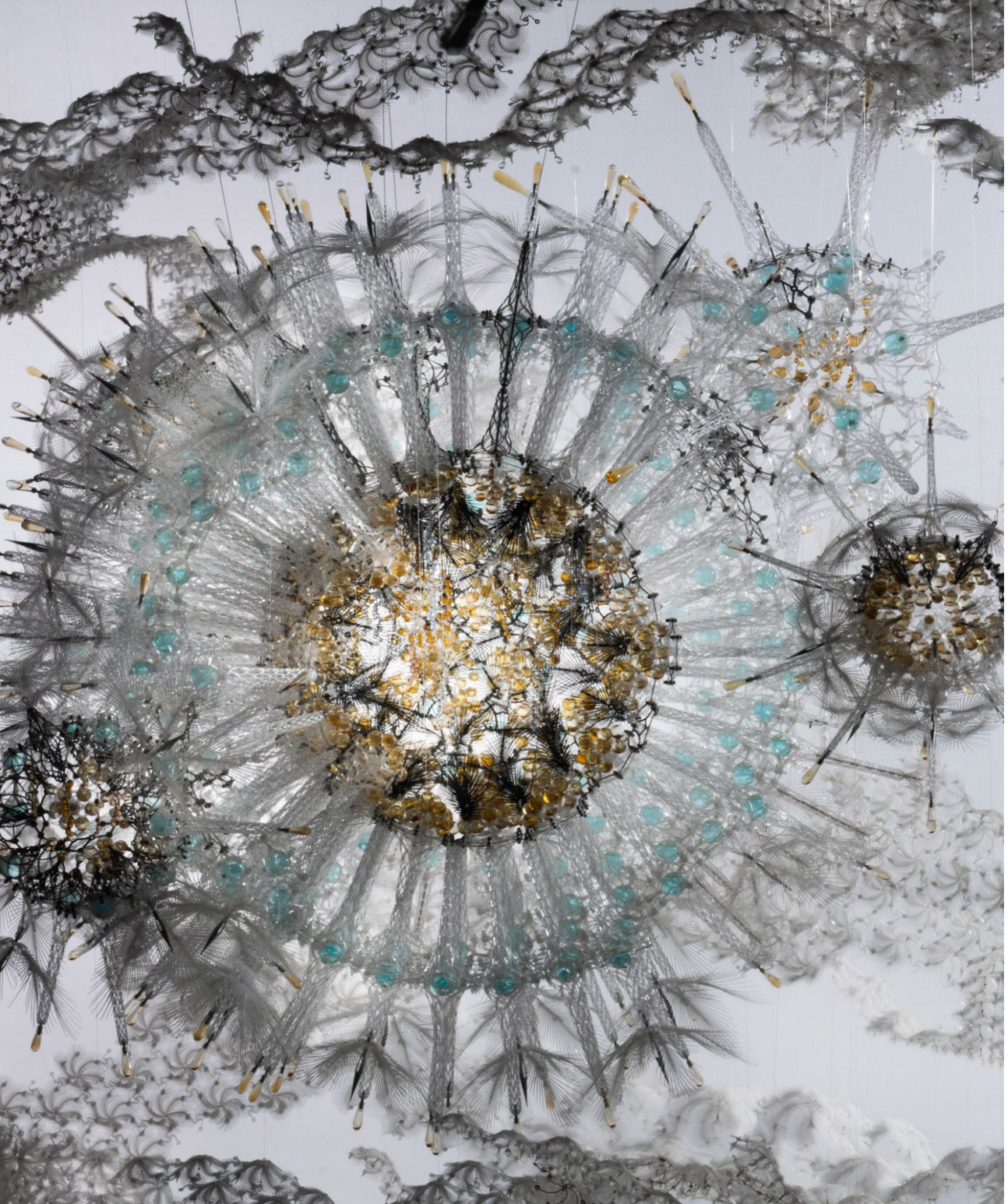
Nicholas Negroponte, Being Digital (New York: Vintage Books, 1995)

Lucy Bullivant, 4dsocial: Interactive Design Environments (London: AD/John Wiley & Sons, 2007)

Neil Spiller, Digital Architecture Now: A Global Survey of Emerging Talent (London: Thames & Hudson, 2009).

Michael Fox and Miles Kemp, Interactive Architecture (Princeton: Princeton Architectural Press, 2009).

Contents

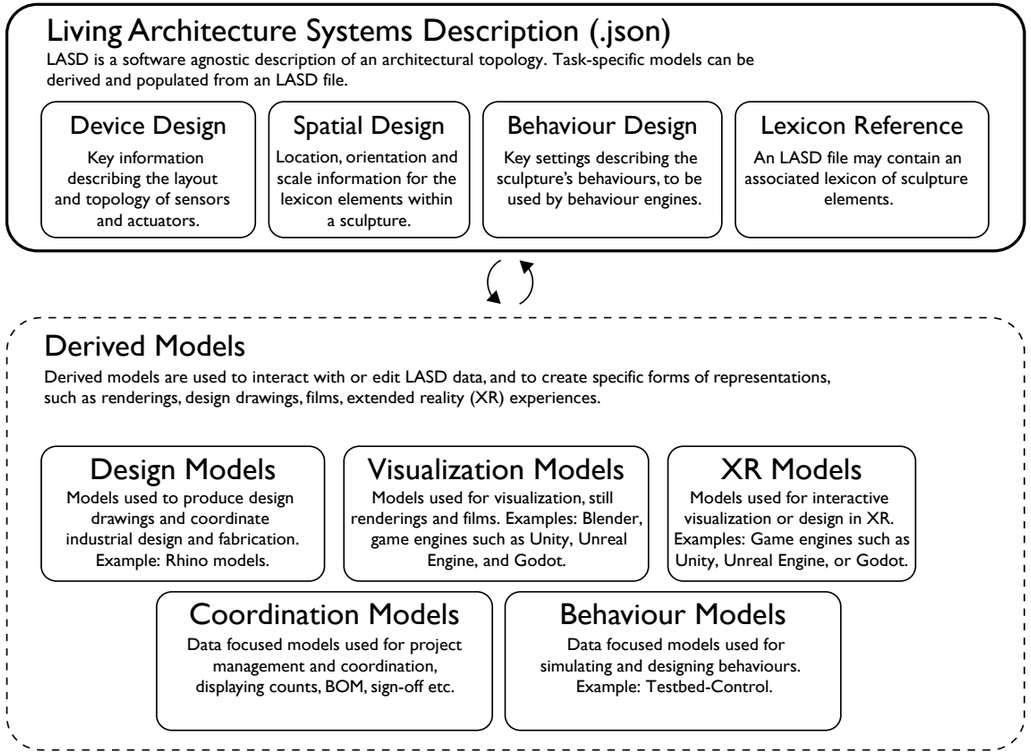


Facing Page
Detail of *Nest* from *Threshold* sculpture group, Philip Beesley/ Living Architecture Systems Group, permanent collection of San José International Airport Art + Technology Program (2021).

Introduction

This publication documents a data framework titled Living Architecture Systems Description that describes the organization of components and assemblies within a series of experimental responsive architectural environments. These specialized environments have been developed by the Living Architecture Systems Group (LASG), an international partnership of researchers, artists, and industrial collaborators studying how we can build living architectural systems. The environments function both as art installations within museums and galleries and, in parallel, as testbeds that support scientific research. The qualities of living architecture systems are being explored in order to develop sustainable, adaptive environments that can move, respond, and learn, and that are inclusive and empathic toward their inhabitants. The Living Architecture Systems Description (LASD) documented here is designed to support computational modelling, time-based media development, and real-time control for these environments.

The LASD framework arose from the need to support integrated workflows for the design and management of evolving components within experimental architectural prototypes. The workflows demand easy translation between widely varying 3D computational modelling tools and real-time simulation environments. This publication provides core descriptions of this framework including background, definitions, and case studies. It offers information on coordinating the LASD with common digital content creation (DCC) tools to facilitate data exchange between the specialized software developed for living architecture projects. Integration with current software modelling platforms includes: Rhino, Blender, and Unreal Engine.



In the intricate world of a living architecture testbed, undulating fabrics and clouds made of interlinking, flexible lattices weave around skeletal shell forms encrusted with glass vessels. Networked sensors and actuators[2] detect sound and movement in both the testbed and surrounding environment and, in response, send ripples of light, motion, and sound through the system. This wealth of complexity gives rise to a form of responsive architecture that begins to take on life-like behaviours.[3] Supporting these behaviours, the design of LASG testbeds is governed by hierarchies of well-defined orders. Components are combined into groups that form assemblies and sub-assemblies. These combine to form various zones that, together, form a living architecture testbed. The LASD provides a framework to describe the hierarchy of these assemblies and components, their physical layout, and the topology of the network of sensors, actuators, and microprocessors that bring a testbed to life.

Above
A high-level overview of the data categories contained within an LASD file and the types of models that can be derived from LASD data.

2 Actuator refers to any device or mechanism that is powered and responds to electronic control.

3 Amy M. Youngs, "The Fine Art of Creating Life," Leonardo 33, no. 5 (2000): 377–80.

The structure of the Living Architecture Systems Description helps to make the complexity of living architecture testbeds accessible. It provides a consistent set of terms that can be used to organize data within the digital design tools used by the teams of the LASG, including the LASG's own Testbed-Control Software,[4] which is used to control dynamic behaviour within living architecture environments. The systems description and its associated toolsets extend user interfaces within these tools, providing a way to label and edit LASG systems data within 3D modelling environments and supporting the transfer of digital models of testbeds between digital design tools.

The LASD enables designers to explore the emergent properties of layered components and systems as they create new interactive and responsive environments.[5] It offers a fundamental "syntax" that can support conceptions of living architecture, enabling designers, thinkers, and builders to work within complex, turbulent environments.

4 *Testbed-Control* is the name of the software suite developed by the Living Architecture Systems Group that operates testbeds' behaviour systems, orchestrating real-time sensor reading and actuator responses.

5 Scott F. Gilbert and Sahotra Sarkar, "Embracing Complexity: Organicism for the 21st Century," Developmental Dynamics: An Official Publication of the American Association of Anatomists 219, no. 1 (2000): 1–9.

Background

Facing Page
Detail of *Aegis* from *Transforming Space: Aegis and Noosphere* testbed sculpture, Philip Beesley/ Living Architecture Systems Group installed at the Royal Ontario Museum, Toronto (2018).

The Living Architecture Systems Description represents the convergence of many years of various and distributed cross-disciplinary workflows used by the Living Architecture Systems Group (LASG) studio to create living architecture testbeds and to model their behaviour. The workflows described in this section have helped integrate the hierarchies and organization found in 3D digital design models of the physical testbeds into the simulation and control systems that test and control their behaviour. systems.

Chronology of Preceding Workflows

Integrated digital models are central to the design to production workflow used in the creation of a living architecture testbed. Integrated models[6] are backed by layered and deeply nested, tree-like data structures that contain source definitions of archetype blocks[7] containing components (e.g., metal spars), subassemblies (e.g., an LED and a hexapod spar assembly), and assemblies (e.g., a sphere unit) and arrayed instances of models,[8] each containing a unique position, orientation, and identifier. The hierarchical organization of instanced copies of defined blocks within a model closely mirrors the practical hierarchies that organize the physical testbeds.

The various layers that make up living architecture environments, such as physical scaffolds, distributed responsive systems, and the non-structural elements (e.g., Mylar fronds and glass vials) that give them their character, are intermixed within different levels of these frameworks. Position and orientation of instance objects are rooted in underlying meshwork geometries that are developed alongside component and assembly blocks. These meta-geometries exist separately from this instancing data structure. Block hierarchies and meta-geometries are utilized in design workflows and documentation as helpful abstractions of complex assemblies to guide physical construction and installation sequence for the physical construction of testbeds.

6 Model in this context refers to an integrated, coordinate-based, 3D digital model of an entire testbed environment.

7 Blocks provide a means to manage repeating data within a model by containing a single definition that can be represented in many copies.

8 Instances are specific occurrences of blocks within a 3D model. All instanced copies of a given block definition are updated each time the geometry that defines the block is edited.

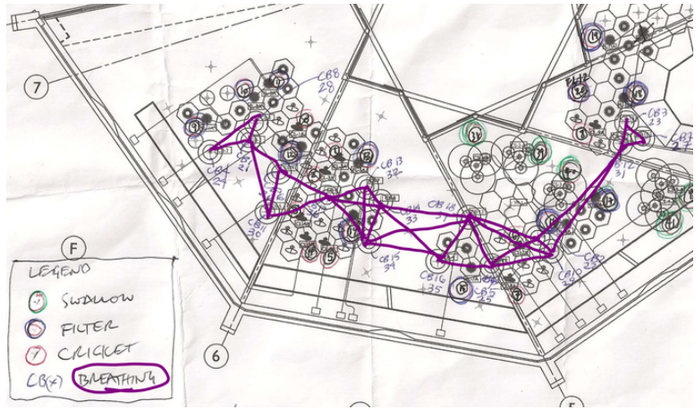
The LASG's Testbed-Control Software manages both a testbed's global logic and local communication between the hardware devices (e.g., Raspberry Pi single board computers and ESP32 microcontrollers) within it. The software architecture follows principles of object-oriented programming.[9] Each type of active physical element within the testbed, such as an actuator or sensor, is modelled as a high-level software class. These classes contain data, such as devices' real-world positions and unique identifiers, and reusable blocks of code, such as routines that describe how actuators behave when subjected to different stimuli. In object-oriented programming, classes are instanced as objects at runtime and can be controlled and manipulated modularly. This object-oriented framework links the physical objects in a testbed to their digital counterparts, laying the groundwork for a simulation environment in which to experiment with behaviours and effects within a testbed that are felt both locally and globally. Translated into a pre-visualization, the simulation allows designers to separate their iterations on a testbed's composition and choreography from the physical structure. This means designers can work in parallel on physical and digital systems before, during, and after an installation is complete. Crucially, this working method relies on trading accurate and concrete spatial information about active devices within a testbed between diverse working teams.

In the testbeds in the Hylozoic Sculpture Series,[10] the design of behaviour systems was based on an abstract model of a testbed's active components and their interrelationships. Software control systems in these earlier generations of living architecture did not rely on behaviour modelling related to specific 3D locations of active devices. Instead, "neighbourhood maps" were included in the preceding versions of firmware of all microcontrollers within a testbed.

These maps were hard-coded lists of groups that described relationships such as spatial proximity and adjacency or common type. Within this framework, the behaviour designer's work was implicitly spatial, and relied on a stable 2D-representation of a testbed environment, but the scripted routines that handled lower-level execution of behaviour systems within and across a testbed's microcontroller network operated at a higher level of abstraction. It was therefore not possible to centrally control, simulate, or visualize testbed behaviour.

9 Object-Oriented Programming is "a type of computer programming in which programs are composed of objects which communicate with each other, which may be arranged into hierarchies, and which can be combined to form additional objects." Merriam-Webster.com Dictionary, s.v. "object-oriented programming," accessed January 13, 2023, https://www.merriamwebster.com/dictionary/objectoriented%20programming.

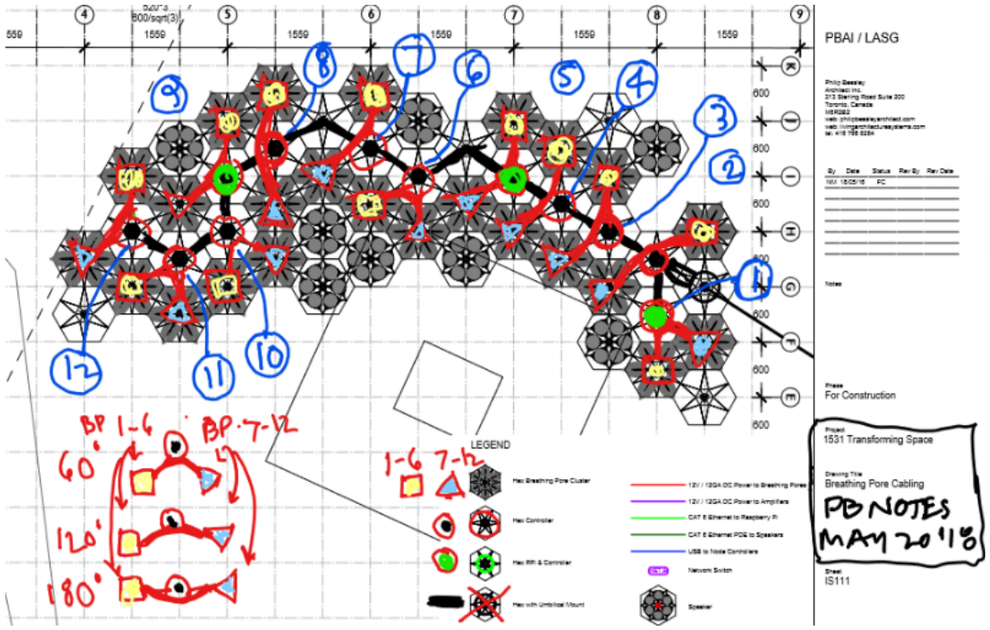
10 This series includes Hylozoic Ground, Canada's entry to the 2010 Venice Architecture Biennale

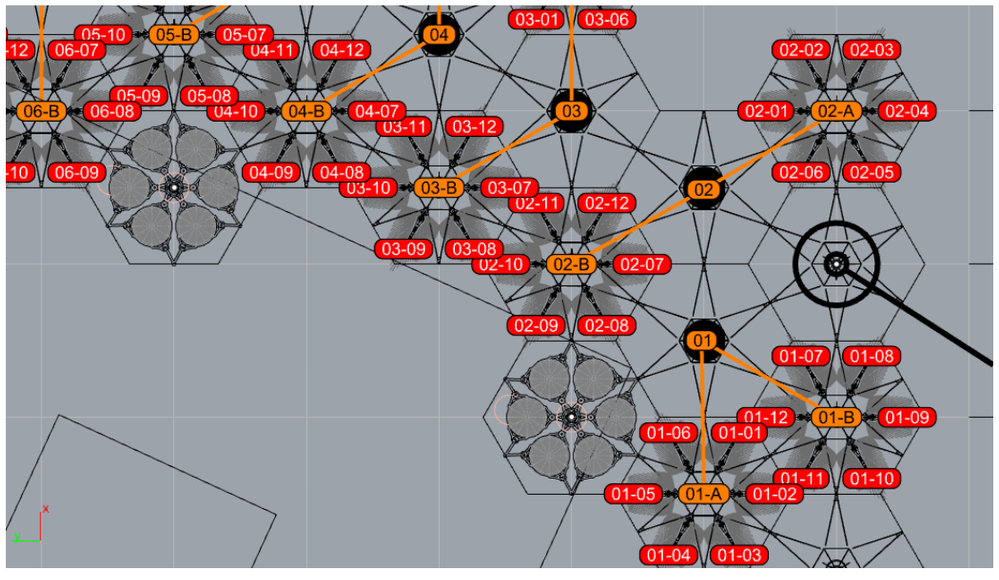


Right
Example of a "neighbourhood map" from the *Hylozoic* sculpture series (2010).

Direct data exchange between design models and the software systems used to control testbeds first emerged in 2018 with the LASG testbed sculpture titled *Transforming Space: Aegis and Noosphere*, installed at the Royal Ontario Museum in Toronto. The digital data exchange that allowed the digital model of this installation to communicate with the software that controlled the project was accomplished using manual data-entry within the workspace of a Rhino 3D model. A simple Rhino object called Text Dot offers a readily usable data structure for storing a string tag linked to {X, Y, Z} spatial coordinates corresponding to a model's World Coordinate System (WCS).

Composition and data-entry within Rhino come with the benefits of working in a visual modelling environment. Data tagged as text dots in Rhino can be precisely placed by a designer working in their native model space.





Facing Page Top
Work-in-progress markup showing device grouping and labelling scheme for Aegis, during installation at the Royal Ontario Museum, Toronto.from the Hylozoic Sculpture Series.

Facing Page Bottom
Screenshot showing Rhino Text Dots used to model a region of Aegis for installation at the Royal Ontario Museum, Toronto.

11 RhinoPython is a scripting interface for Rhino that exposes programmatic read and write access to 3D model data.

12 RhinoScriptSyntax allows RhinoPython to provide access to hundreds of Rhino commands through an easy-to-use Python wrapper. See more at: "Rhino. Python Guides," Rhino Developer, Robert McNeel & Associates, accessed January 19, 2023, https://developer.rhino3d.com/guides/rhinopython/.

Right
Truncated table view of CSV contents output from the RhinoPython script shown above, and additional data about devices that was added after.

13 The labelling scheme used here evolved from the physical labelling effort in the directly preceding testbed installation, *Amatria*, at Luddy Hall, Indiana University in Bloomington. The installation team installed hang tags in Amatria adjacent to each of its hundreds of actuators. Each tag contained information about upstream hardware devices. This labelling was based on a paper-based "Device Locator" binder that contained detailed information about all installed hardware, including controllers' identifiers and downstream.

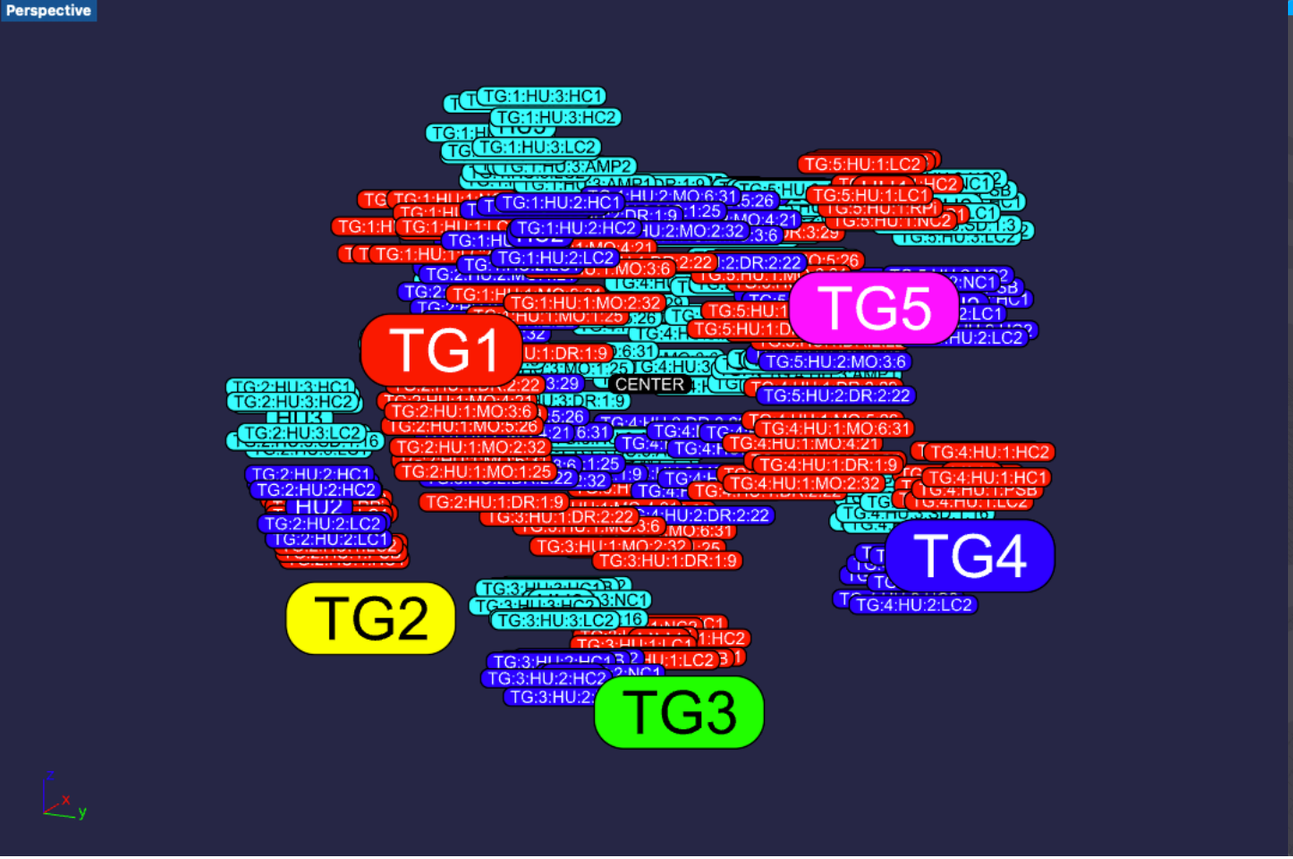
Programmatic access to objects and their attributes through RhinoPython [11] using RhinoScriptSyntax [12] allowed the data tied to text dots to be exported from a Rhino 3D-modelling environment to a simple plaintext format (Comma-separated value, or .csv file) parseable by the software controlling the testbed:

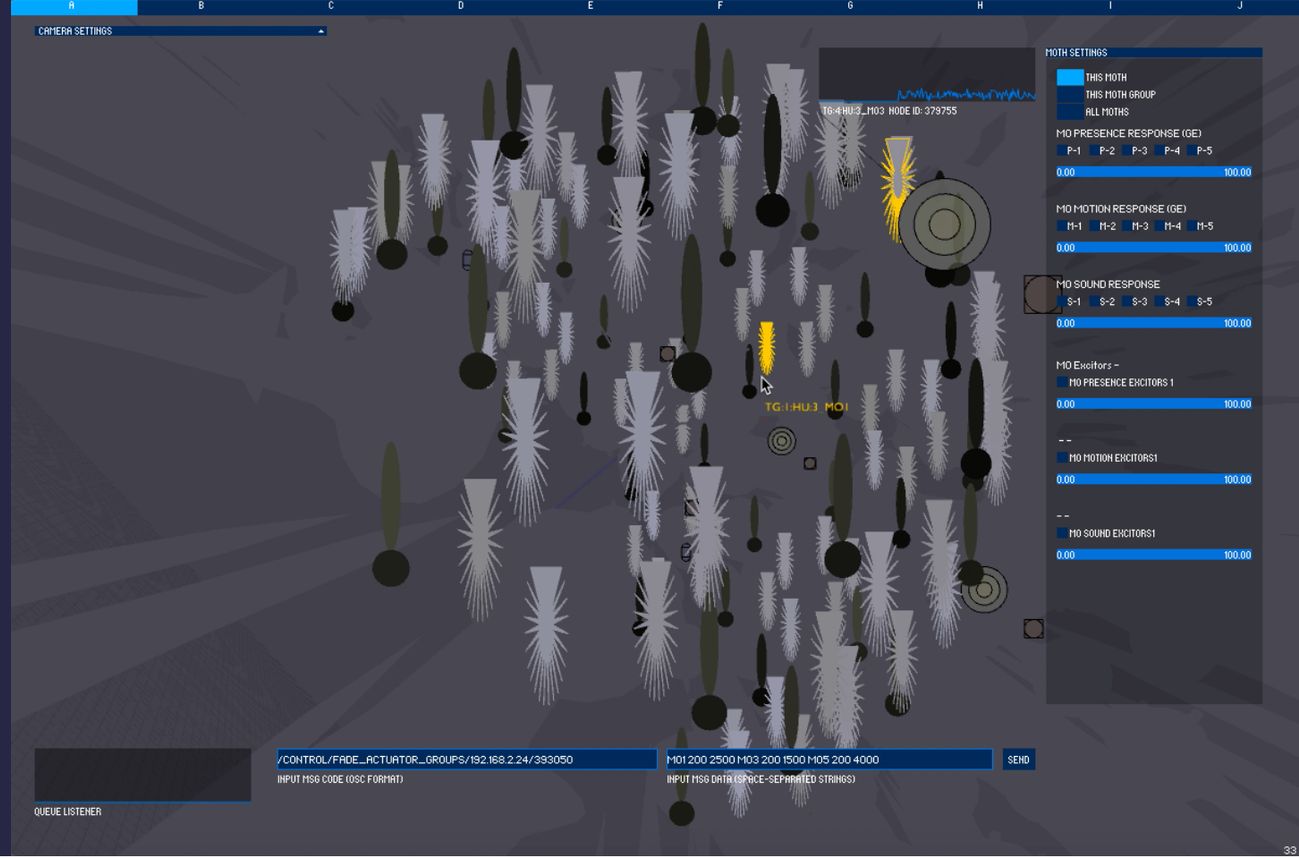
```python
import rhinoscriptsyntax as rs
import csv

textDots = rs.GetObjects(message = "Select Text Dots" , filter = 8192)
dir =  "C:\\User\\PBSI\\Documents\\"

with open(dir +  "devicelocator.csv" ,  'wb' ) as f:
    writer = csv.writer(f, delimiter =  ',' )
    writer.writerow([ "ID" ,  "x" ,  "y" ,  "z" ])
    for textDot in textDots:
        objectID = rs.TextDotText(textDot)
        point = rs.TextDotPoint(textDot)
        line = [objectID, point.X, point.Y, point.Z]
        writer.writerow(line)
```

ID	x	y	z
'01-A	2701.36	-3759.00	7809.92
'01-01	2839.93	-3759.00	7569.92
'01-02	2978.49	-3759.00	7809.92
...			
'01-B	3220.98	-3759.00	7509.92
'01-07	3082.41	-3759.00	7269.92
'01-08	3359.54	-3759.00	7269.92
...			

Section	Peripheral	IP	UID	Pin	Type
BP1-1	IR1-1	10.14.4.206	335496	A0	sensor
BP1-1	SMA1-1-1	10.14.4.206	335496	3	actuator
BP1-1	SMA1-1-2	10.14.4.206	335496	4	actuator
...					
BP1-2	IR2-1	10.14.4.206	335496	A1	sensor
BP1-2	SMA1-2-1	10.14.4.206	335496	25	actuator
BP1-2	SMA1-2-2	10.14.4.206	335496	32	actuator
...					

Rhino-based data for *Aegis and Noosphere* followed a transitional labelling convention, leading to the systematic organization that has been adopted in the syntax of the Living Architecture Systems Description. In the preceding organization, simple two-part labels contained a *group* number that corresponded to a physical control node within the sculpture, and a *device* number that corresponded to an addressable port number on the hardware controller.[13] This labelling scheme did not provide an end-use repository of testbed device information, but it did provide enough information for software designers to evaluate conditional statements and use lookups in order to expand the data to an appropriate level of detail for use in control software.





Development of the object-oriented Testbed-Control Software began in late 2018, with the goal of constructing a virtual software environment closely related to new physical testbeds that would allow for simultaneous simulation and control. This new development answered the practical need for this central software system to be able to communicate with downstream microcontrollers, and demanded a more rigorous and extensible naming standard compared to the conventions used for previous generations of testbeds. A new naming scheme was developed to accommodate various different types of nested assemblies and their related hardware device connection topologies. This naming scheme categorizes elements hierarchically into Group, Node, and Device types, each numbered serially corresponding to their parent groups.

Above
A screenshot showing Rhino Text Dots used to model *Noosphere* for installation at Futurium, Berlin (2019).

Facing Page
A screenshot of an early version of Testbed-Control software showing a simulation of the Futurium Noosphere installation. Device names are displayed when selected.

The Testbed-Control Software debuted with Futurium Noosphere in 2019 in Berlin. This testbed comprises a clustered group of shell structures that form interlinked spherical meshwork scaffolds. High-power LED lights, vibrating fronds, and high-fidelity omnidirectional speakers were integrated within the primary double-shell sphere scaffold, along with proximity and motion sensors and sound detectors. When viewers wave and gesture, a distributed system of some two dozen microcontrollers works in concert with a central control computer in order to orchestrate intricate patterns of rippling light, sound, and motion. The simplified structuring of groups, nodes, and devices within this testbed provided an opportunity to launch this new software and data exchange protocol.

Precedents

An LASD file adopts the *JSON* electronic data interchange format. JSON (JavaScript Object Notation) is an open standard filetype that uses key–value pairings to store data of various types. In this file format, data is stored in plain text and is easy for both machines to parse and generate and for humans to read and write.[14] LASD has been designed to support interoperability between different software packages; it is software agnostic. Open source frameworks for 3D data interchange such as Universal Scene Description (*USD*), developed by Pixar Animation Studios, and GL Transmission Format (*glTF*), developed by Khronos Group, have served as valuable precedents and have informed the development of LASD.

During early development of LASD, core principles of USD and glTF helped form the guidelines that would establish the ethos of the LASD. Developed by Pixar Animation Studios starting in 2012, USD is a software platform designed to allow for the interchange of assets between Pixar's own variety of internally-developed digital content creation (*DCC*) tools. It was first published in 2016 as open source software. The USD platform covers domains such as 3D geometry, shading, lighting, and physics. Pixar states three goals to be addressed by USD's ongoing development:

- Provide a rich, common language for defining, packaging, assembling, and editing 3D data, facilitating the use of multiple digital content creation applications.
- Allow multiple artists to collaborate on the same assets and scenes.
- Maximize artistic iteration by minimizing latency.[15]

LASD's file syntax and structure relates to the glTF file format. Khronos Group has developed glTF as a format to bridge the gap between 3D content creation tools and modern graphic language applications by providing an interoperable format for the transmission and loading of 3D content.[16] Khronos states their motivation with glTF is to provide a file transmission format that can be "loaded and rendered with minimal processing."[17] The first part of a glTF file is a core scene description represented in JSON format describing relationships and hierarchies of assets within the file. Assets such as 3D meshes and 2D textures and images are stored in binary data buffers to minimize file size. In the glTF format specification, Khronos states the following design goals:

- Compact file sizes
- Fast loading
- Runtime-independence
- Complete 3D scene representation
- Extensibility[18]

Despite being represented in part in JSON, Khronos considers human-readability to be outside the scope of glTF design goals

While the LASD is more limited in scope than USD and glTF, similar goals are defined to guide ongoing development of this framework:

1. **The LASD schema should be software agnostic:** 'Software agnostic' denotes the ability to work with a wide variety of design tools. Though the LASD is primarily used for the transfer of model data between specific DCC tools such as Rhino and Blender, its structure should remain open and easily adoptable, avoiding the use of proprietary encodings or file formats within its core schema. Where parts of the LASD workflow do incorporate proprietary encodings or file formats (such as the Lexicon Mesh Library, which uses the file format *fbx*), these should operate separately from and without affecting its core functionality.
2. **The LASD schema should be extensible:** As design lexicons and libraries of behavior systems continue to evolve, the LASD framework should be extensible to accommodate new systems and new data.
3. **An LASD file should be human readable:** As much of the data in an LASD file should be stored in a manner that can be read and understood at a high level by a person working with 3D modeling or software and behavior design of a testbed. JSON key–value pairs are the preferred format for storing data, allowing nested, lower-level data to be folded into simple, higher-level contextual String identifiers.
4. **An LASD file should be lightweight:** An LASD file should contain the data necessary to describe hierarchies, topologies, and systems that comprise a living architecture testbed. Storage of large bodies of supporting data that is not human-readable, such as mesh files, should be stored outside the LASD file in a separate Lexicon directory.

14 "Introducing Json," JSON, accessed January 13, 2023. https://www.json.org/json-en.html.

15 Introduction to USD: Why Use USD?," Introduction to USD - Universal Scene Description 22.11 documentation, accessed January 14, 2023. https://graphics.pixar. com/usd/release/intro.html#whyuse-usd.

16 "GITF/Readme. md at Main · KhronosGroup/GLTF: Introduction," KhronosGroup, GitHub, accessed January 14, 2023. https://github.com/KhronosGroup/glTF/blob/main/specification/1.0/README.md#introduction.

17 "GITF/Readme. md at Main · KhronosGroup/GLTF: Motivation," KhronosGroup, GitHub, accessed January 14, 2023. https://github.com/KhronosGroup/glTF/blob/main/specification/1.0/README.md#motivation.

18 "GITF/Readme. md at Main · KhronosGroup/GLTF: Motivation," KhronosGroup, GitHub, accessed January 14, 2023. https://github.com/KhronosGroup/glTF/blob/main/specification/1.0/README.md#motivation.

Facing Page
Photograph of *Noosphere* testbed, Living Architecture Systems Group, installed at *Futurium*, Berlin (2019).

Organization of an LASD File

The LASD description of a testbed comprises two primary parts: the LASD data and the Lexicon Mesh Library. The Lexicon Mesh Library contains the detailed mesh representation of each of a testbed's components (currently stored as either an .fbx or .gltf file), exported from a 3D-modelling software such as Blender or Rhino. The LASD data file is a text file that stores information about the physical parts and device topologies that make up the testbed.

LASD data is stored using JavaScript Object Notation (JSON, or .json). A JSON object consists of a dictionary of key value pairs, where each key can be used to look up its corresponding value. A typical LASD .json file consists of four top-level keys, Metadata, Lexicon, Instances, and Devices, each of which contains its own subset of data that, all together, serve to completely describe the physical structure and actuated elements of a living architecture testbed.

Units and Coordinates

The LASD uses left-handed, Z-up, coordinate space. Distances are defined in millimeters, and rotation angles are defined in radians.

Metadata

The Metadata key contains general information that pertains to the file or the testbed as a whole. Information such as the last revision date, file author, and project name are stored here.

Instances

The concept of instances forms the core of the LASD's data structure. An instance in LASD is a specific occurrence of a defined type or class of object. This is similar to the term's definition in the object-oriented programming (OOP) framework. Each instance has its own unique properties, such as its local position and rotation. Shared properties that are common across all instances of an object type, such as shape and size, are referenced from the object definition and are not stored individually within instances.

The Instances key in the LASD file contains a sub-dictionary of all top-level instances that comprise a sculpture. Each instance contains information about its name, layer, and position within the sculpture, as well as a reference to the particular Lexicon definition that defines the common properties of that instance type.

Lexicon

The lexicon contains the definitions of all the object types that comprise a sculpture. Each object within the lexicon serves as a definition of all of the common properties that are shared across all instances of that type. An LASD lexicon is divided into two categories: components and assemblies.

Components

Components are the primary units of the LASD. Each component represents a single element of a living architecture testbed. In its simplest form, a component definition would declare only a unique name for itself; however, components often contain additional data such as primitive representational geometry and detailed mesh models. Further data, such as a device type and device ID, may be provided if the component represents an actuator or sensor device.

Commonly used component definition data is exposed in the user interface by the LASD import plugins for Rhino and Blender, allowing designers and modellers to easily input data pertaining to individual components natively within model workspace. These interfaces also include the ability to add user-defined key value pairs as needed.

Assemblies

Assemblies represent groupings of lexicon elements. Like components, assemblies have a name, primitive geometry, and can contain additional device data and user-defined data; however, instead of a mesh representation, assemblies contain a dictionary of "Nested Instances." Like a top-level instance, a nested instance is a specific occurrence of a lexicon object containing information about its unique properties, such as position and rotation, and a reference to its lexicon definition. Nested instances store their positions relative to the origin point of the assembly, rather than relative to the global space of the testbed. Assemblies allow a testbed to be described as a hierarchy of repeated elements, eliminating the need to store the positions of every single component present within it.

Assemblies are analogous to blocks in Rhino or collection instances in Blender.

1

```
“GLASS_Dissapative_Tip” : {
 “name” :  “GLASS_Dissapative_Tip” ,
},
```

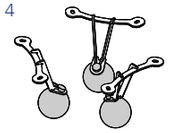
2

```
“GLASS_Ball_Large” : {
 “name” :  “GLASS_Ball_Large” ,
},
```

3

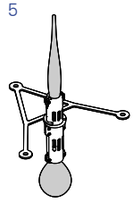
```
“PLATE_X” : {
 “name” :  “PLATE_X” ,
 “is_connector” : true,
},
```

4



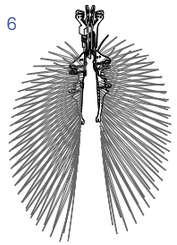
```
“GLASS_Hexapod” : {
 “name” :  “GLASS_Hexapod” ,
},
```

5



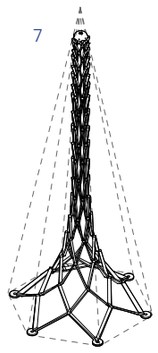
```
“ACTUATOR_Double_Rebel_Star” : {
 “name” :  “ACTUATOR_Double_Rebel_Star” ,
 “collection_type” :  “DEVICE” ,
 “device_type” :  “actuator” ,
 “device_current_type” :  “HC” ,
 “device_id” :  “DR” ,
}
```

6



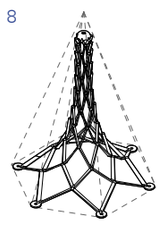
```
“ACTUATOR_Moth” : {
 “name” :  “ACTUATOR_Moth” ,
 “collection_type” :  “DEVICE” ,
 “device_type” :  “actuator” ,
 “device_current_type” :  “HC” ,
 “device_id” :  “MO” ,
}
```

7



```
“SPAR_Long_Acrylic” : {
 “name” :  “SPAR_Long_Acrylic” ,
 “primitive_type” :  “CONE” ,
 “primitive_radius” :  “150” ,
 “primitive_height” :  “650” ,
 “primitive_sides” :  “6” ,
},
```

8



```
“SPAR_Short_Acrylic” : {
 “name” :  “SPAR_Short_Acrylic” ,
 “primitive_type” :  “CONE” ,
 “primitive_radius” :  “150” ,
 “primitive_height” :  “350” ,
 “primitive_sides” :  “6” ,
},
```

Components & LASD Descriptors

1 Dissipative Tip 2 Ball Glass 3 X Plate 4 Hexapod 5 Double Rebel Star
6 Moth 7 Long Spar 8 Short Spar

1

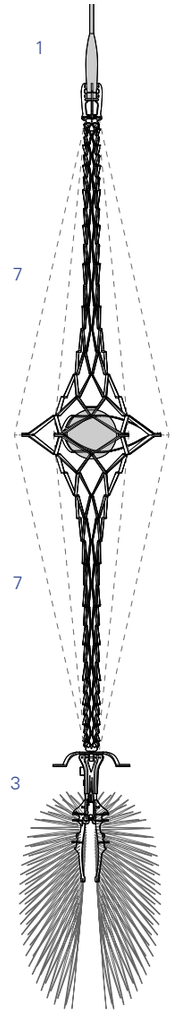
2

7

7

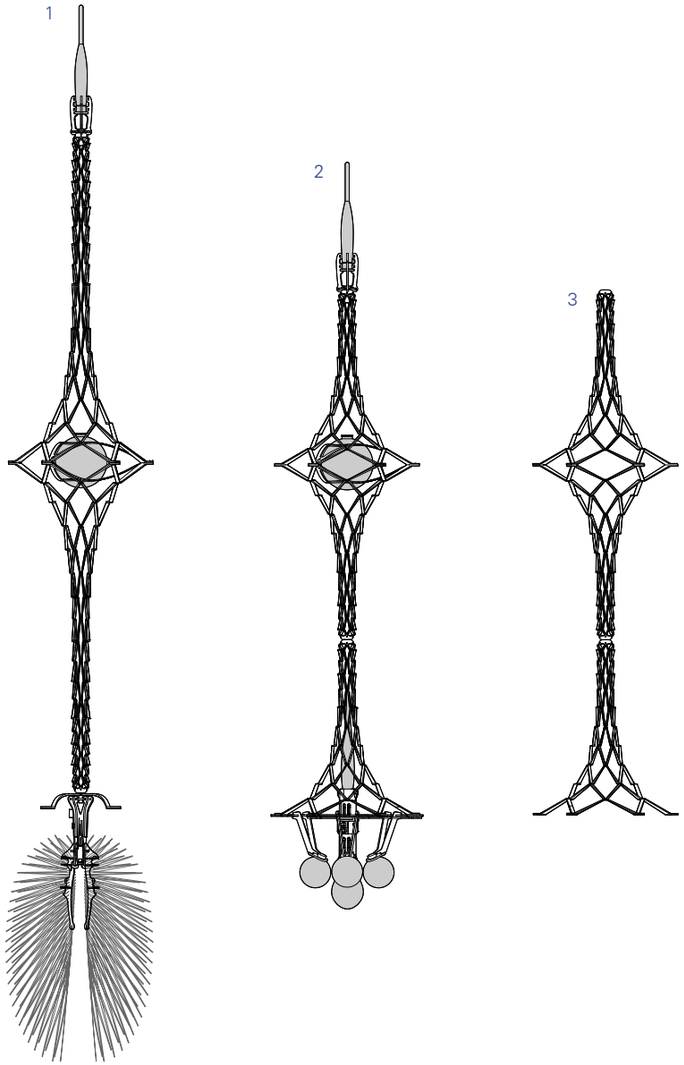
3

6



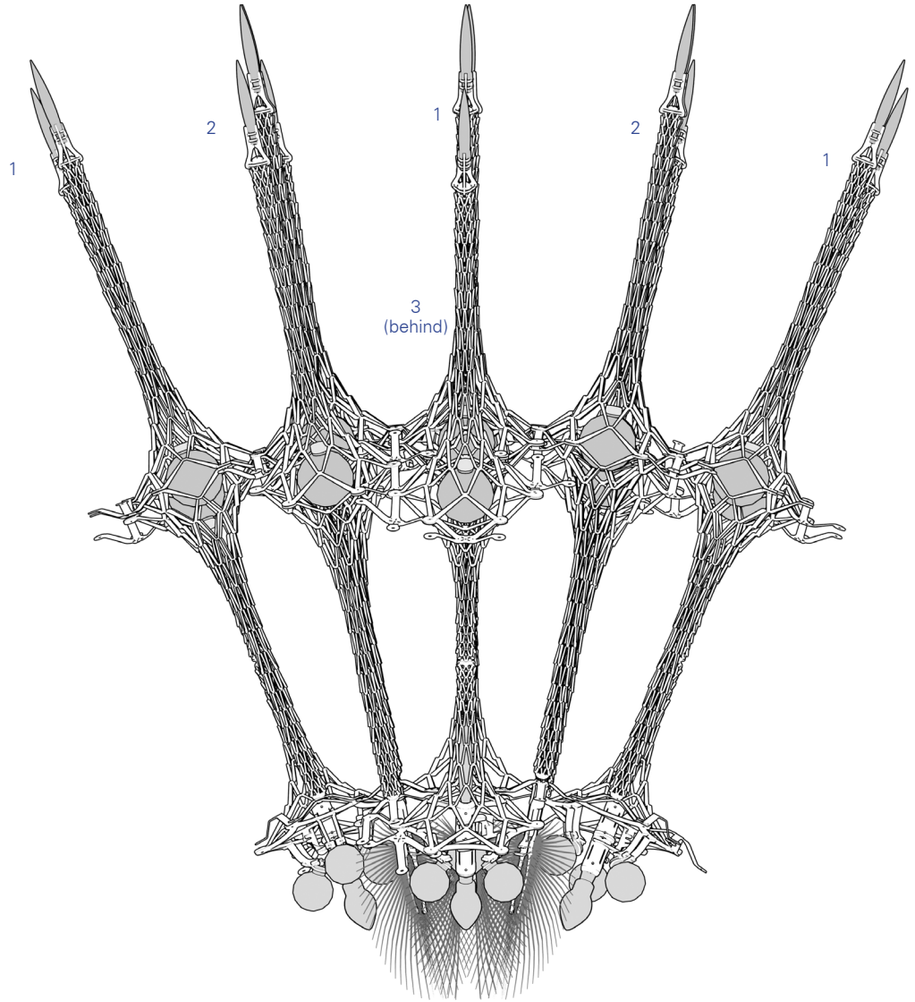
```
MACRO_SPAR_ASSEMBLY” : {
  “name” :  “MACRO_SPAR_ASSEMBLY” ,
  “count” : 1,
  “nested_instances” : {
  “SPAR_Long_Acrylic” : {
  “name” :  “SPAR_Long_Acrylic” ,
  “instance” :  “SPAR_Long_Acrylic” ,
  “layer” :  “MACRO_SPAR_ASSEMBLY” ,
  “location” : [0.0, 0.0, 1.26],
  “rotAxis” : [0.0, 0.0, 0.0 ],
  “rotAngle” : 0.0,
  “scale” : [ 1.0, 1.0, 1.0],
  “matrix” : [
         [1.0, 0.0, 0.0, 0.0],
         [0.0, 1.0, 0.0, 0.0],
         [0.0, 0.0, 1.0, 1.26],
         [0.0, 0.0, 0.0, 1.0]
         ]
  },
  “GLASS_Dissapative_Tip” : {
   ...
  },
  “SPAR_Long_Acrylic” : {
   ...
  },
  “SPAR_Long_Acrylic.001” : {
   ...
  },
  “GLASS_Ball_Large” : {
   ...
  },
  “PLATE_x” : {
   ...
  },
  “ACTUATOR_Moth” : {
   ...
  }
 },
},
```

Spar Assembly & LASD Descriptor



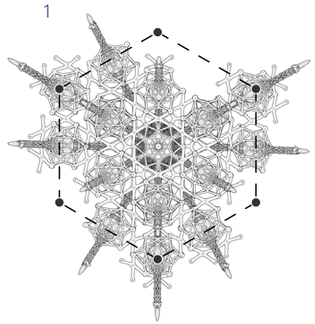
Double Shell Sphere - Spar Assemblies

1 Moth Spar Assembly 2 Rebel Star Spar Assembly 3 Core Spar Assembly

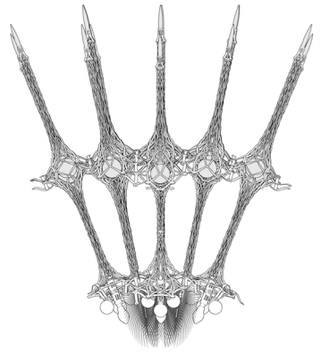


Double Shell Sphere - Sphere Unit Assembly

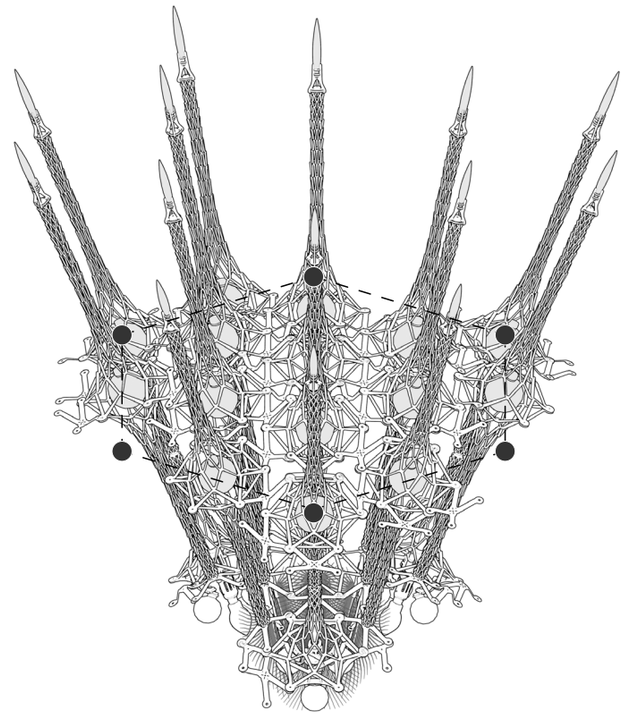
1



2



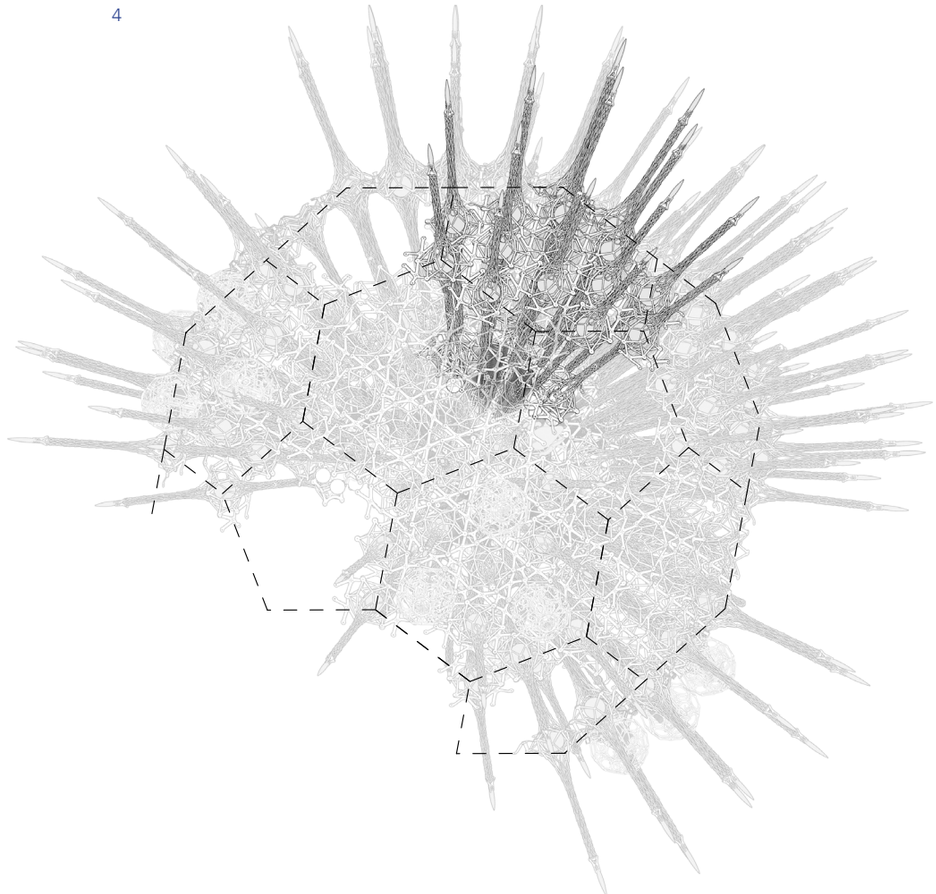
3



Double Shell Sphere - Sphere Unit Assembly

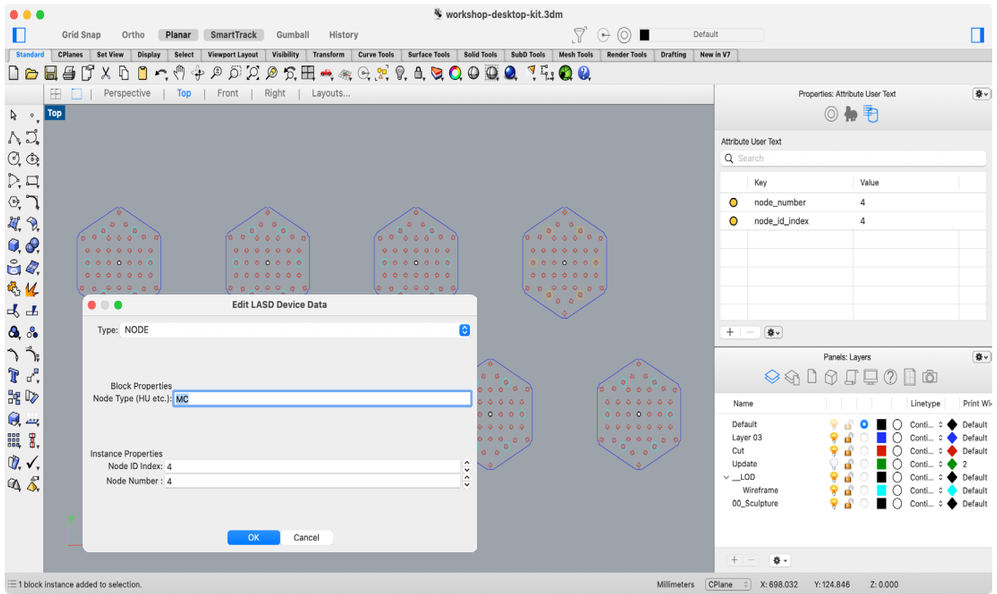
1 Sphere Unit - Top View 2 Sphere Unit - Side View 3 Sphere Unit - Axonometric View

4



Double Shell Sphere - Full Assembly

4 Double Shell Sphere



Devices

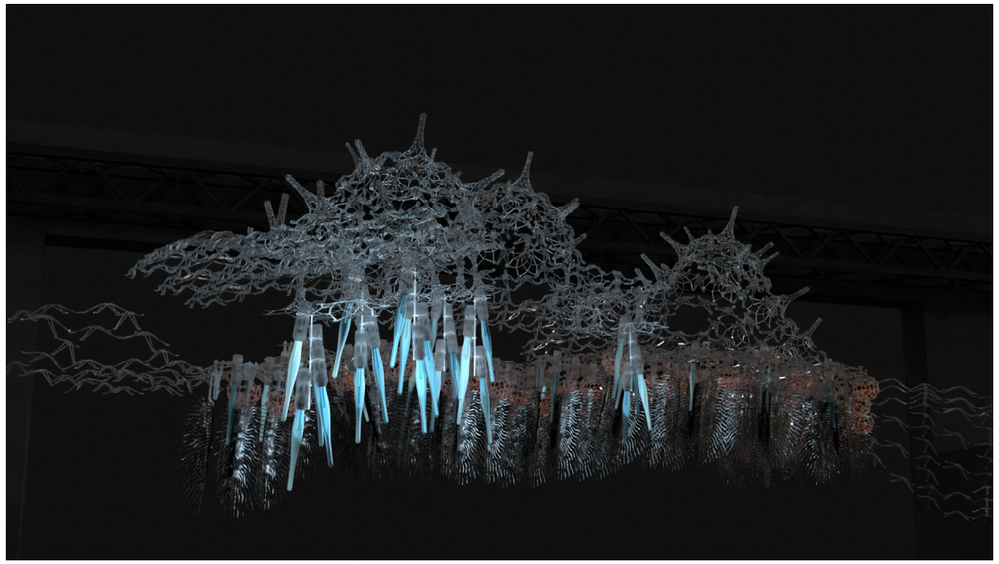
The Devices key contains an array of all device data contained in an LASD file. It is formatted to be easily parsed by Testbed-Control.

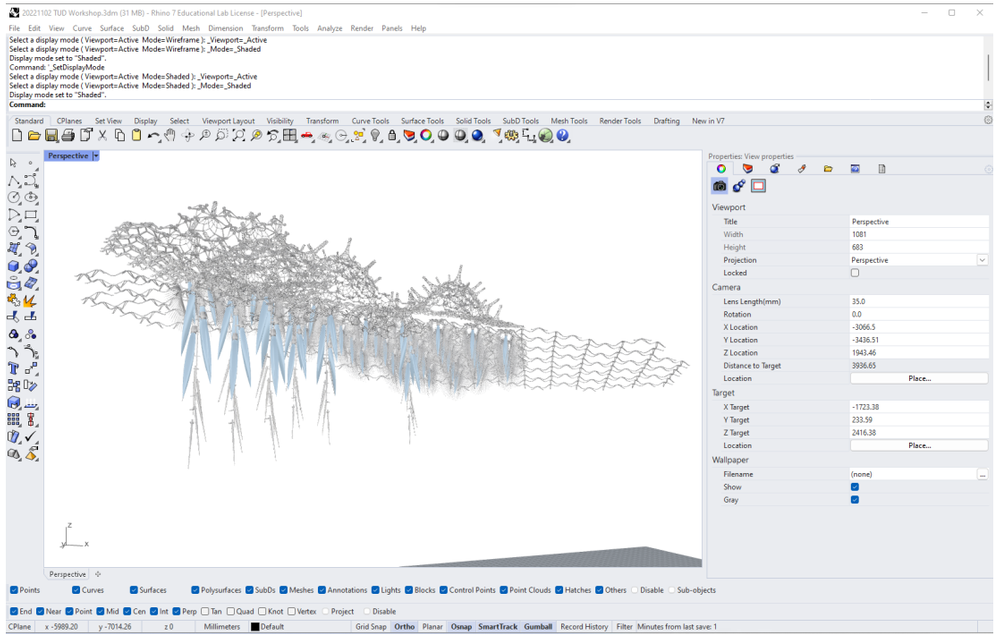
The information contained in the devices key of an LASD file comes from data that is input within the relevant instances (for unique properties) and lexicon objects (for shared properties) within modelling software. The data in the devices key is a flattened representation of hierarchical information and is similar to the way device data was stored in the previous Device Locator CSV files. It serves as a simplified representation of devices' essential location and identification data that is easier to parse in Testbed-Control software.

Above
A desktop array of nodes and actuators uses an LASD file to establish a spatial configuration within Testbed-Control software. This allows designers to develop behaviour and influence profiles using the same hardware that is installed within testbed sculptures.

Facing Page
Device data is entered in a dialog window exposed by the *DeviceLASD* Rhino command, or through Rhino's *Attribute User Text* panel.

The Devices key is always regenerated from the instance and lexicon properties when exporting an LASD file from a digital design tool. Edits to device data should be done only in a digital design tool such as Blender or Rhino, and not directly in the .json text, to ensure data remains synchronized across lexicon and device sections of the LASD file.





Case Studies

Facing Page
The LASD allowed using Rhino and Blender in parallel to develop cycles of design and visualization, respectively, for the design of *Poietic Veil* testbed sculpture, installed at TU Delft Science Centre (2022).

Top A rendering of *Poietic Veil* made using Blender.

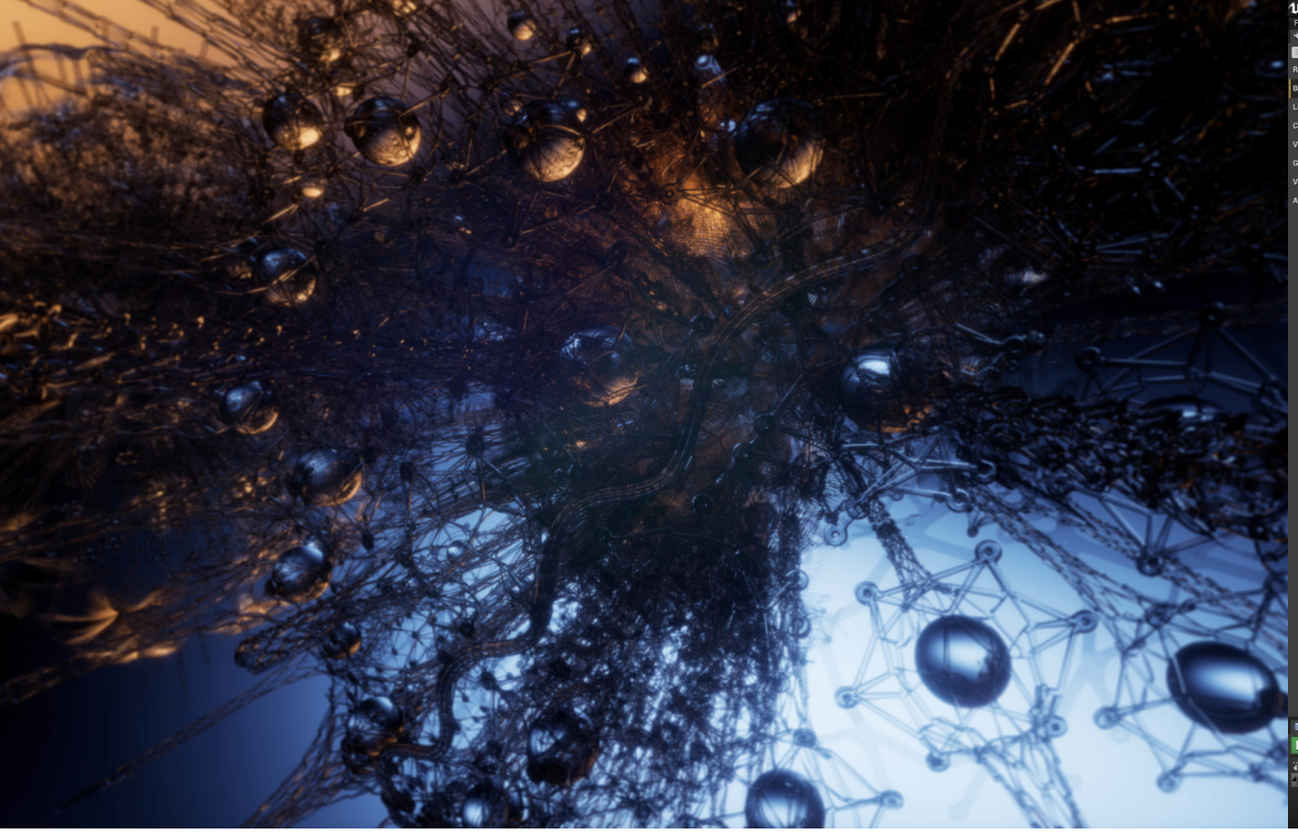
Bottom *Poietic Veil* working model shown in the Rhino modelling environment.

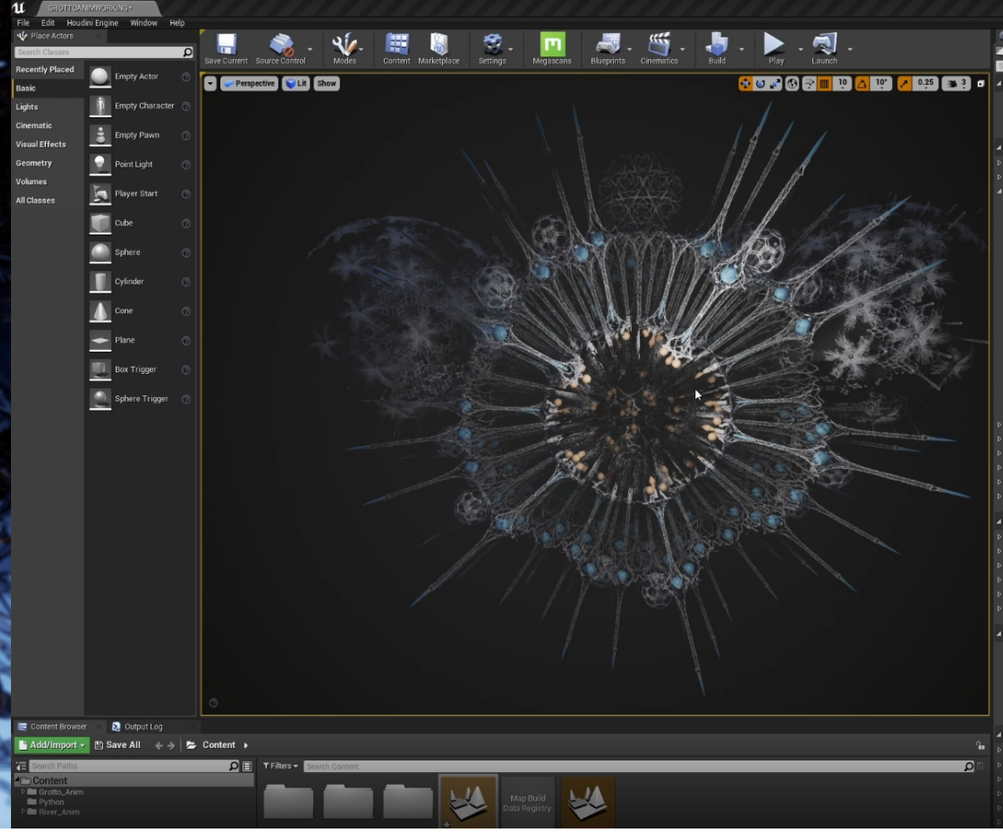
Uses within the Design Studio

The LASD was initially conceived as a set of protocols and procedures that would relieve two main preceding bottlenecks in LASG testbed design workflows. The first of these relates to device data exchange. Previous processes of exchanging testbed device data between digital design tools and LASG bespoke Testbed-Control software were subject to strain each time a working model was updated. LASD would relieve this by exposing new data-input interfaces in the digital design tools used by the LASG and by standardizing the format of testbed device data and providing import-export routines for diverse software packages.

The second bottleneck was the preservation of block-instance organization in model exchange between Rhino and Blender. Exchanging model data through conventional processes that are supported by these two tools often flattens structured data hierarchies. These transfers would result in models that are lack key organization and may result in loss of detailed information. The LASD, in combination with a Lexicon Mesh Library or equivalent Rhino Block Library,[19] allows compositional models to be easily and repeatedly imported and exported between various digital design tools in a non-destructive way. This means designers can, for instance, develop multiple cycles of industrial and architectural design in Rhino, and in parallel develop multiple cycles of lighting, animation, and rendering in Blender, moving efficiently back and forth between those two tools.

19 Lexicon Mesh and Rhino Block Libraries are collections of external files in formats such as .fbx and .3dm, respectively, that correspond to the objects described within the LASD data's Lexicon section. These 3D model files allow programmatic construction of holistic models that respect the structures described within the LASD file. LASD's organization relates strongly to Rhino's block structure.





Animated Digital Media Creation: Cradle Film

During the production of the short film Cradle,[20] developed for the Grove installation at the 2021 Venice Architecture Biennale in collaboration with Warren du Preez and Nick Thornton Jones, the Living Architecture Systems Description was used to facilitate the transfer of high-resolution model files from Blender to Unreal Engine. An LASD importer was created using Unreal's Python API. These exchanges helped integrate the Lexicon Mesh Library into the LASD workflow as a way of linking LASD data to detailed mesh representations exported in conventional file formats.

Above

Still from *Cradle*, Philip Beesley, Warren du Preez, Nick Thornton Jones.

Facing Page

The core grotto model used for *Cradle*, shown in Unreal Engine imported using the LASD.

20 For additional information on *Cradle*, see the folio linked at LASG.ca

Overleaf

Still from *Cradle*, Philip Beesley, Warren du Preez, Nick Thornton Jones.

The LASD importer in Unreal Engine imports testbed data as static and skeletal mesh actors within Unreal and organizes these actors according to the hierarchies defined in the LASD. It also supports the exchange of animated components, with animation data being contained within an fbx file in the Lexicon Mesh Library.

Translating Between Design- and Control-space: Futurium Device Locator

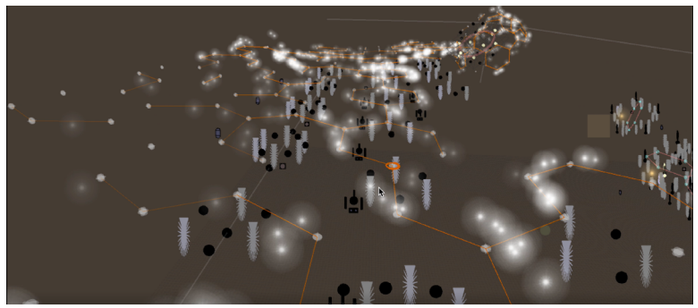
With the spatial aspects of a living architecture testbed well- represented in the LASD and tested across a variety of platforms, the next step in development was to expand the schema to incorporate information about the sensors, actuators, and control systems that bring animated light and motion to a testbed. This prompted development that would expose an interface to Blender and Rhino users for device data entry. Each component that represents a device within a testbed would carry device-type and identification data.

In living architecture testbeds, sets of devices are typically grouped under a single node controller, a microprocessor that manages messaging data and computes an actuator's behaviour. Assemblies of components with device data can be tagged as nodes to be included in the devices data section.

Above
Screenshot from Testbed-Control software showing *Futurium Noosphere*. The device data and primitive geometry shown in this user interface are modelled from the imported contents of an LASD file.

Instances of devices and nodes each also expose their unique per-instance properties, such as their unique identifier (UID) and microcontroller pin assignment, which provides an integer number that corresponds to an addressable port on the hardware device.

On export, the hierarchical data of the LASD's components, assemblies, and instances is processed and flattened into a data structure that more closely matches the studio's previous method of documenting device data, the Device Locator CSV file.



Right
Virtual GridRunner particles are pictured traversing the meta-geometry that underlies the testbed *Meander's* 'river' construction. These virtual particles create spatial patterns of light and vibration in the testbed as they pass the location of actuator objects in the Testbed-Control simulation environment.

Some behaviour systems in development by the LASG studio for testbeds also require additional spatial data, or *meta-geometry*: model elements that do not represent physical objects but enable the spatialization of behaviour controls. For example, the *GridRunner* particle system overlays a virtual grid of vertices within the space of a testbed that is related to and derived from the underlying geometries appearing within the testbed. These geometries do not otherwise appear within nor are they described by the LASD. LASD component definitions allow a user to extend a custom key to declare a *GridRunner Vertex* lexicon component. Instances of GridRunner Vertex define the meta-geometry of the GridRunner network. Once that instance type and network is defined, the objects can be modified in the 3D model space and exported to an LASD file that can be parsed by Testbed-Control's behaviour engine.

The resulting LASD file can be read into Testbed-Control to populate a simulated scene with appropriate sensors and actuators, which can then be used to develop and simulate behaviours and responses to external stimuli. Testbed-Control also reads the primitive data from the component definitions in order to reconstruct a low-resolution preview of the sculpture within the control software.

Design Software Explorations: Polygonal Lattice Tool

By providing a stable framework that allows for the simple exchange of model data between software packages, LASD makes it possible for the LASG to explore designing testbeds outside of existing digital design tools including the Living Architecture Polygonal Lattice (LAPL), a custom design tool.[21] LAPL allows for point-and-click sketch-modeling and physical simulation of the polygonal lattices that form the base geometry of many living architecture testbeds.

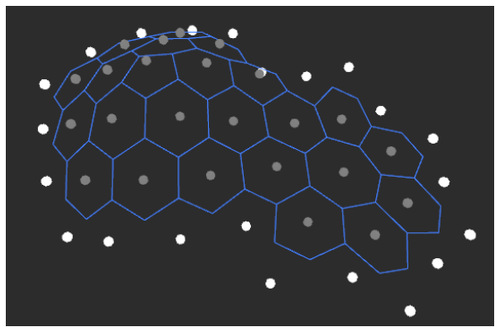
When an LASD lexicon is loaded into the LAPL design tool, the LAPL is able to populate LASD instances on LAPL primitive geometry. The LAPL software selects suitable objects to populate each cell of the polygon lattice using a series of algorithms that evaluate allowed adjacency conditions. The LAPL can also make use of additional lexicon information, such as a species key to ensure, for example, that primary polygonal components (e.g., spars or sargasso cloud cells), instanced on mesh faces, are compatible with their related connectors, instanced on mesh vertices or mesh edges.[22]

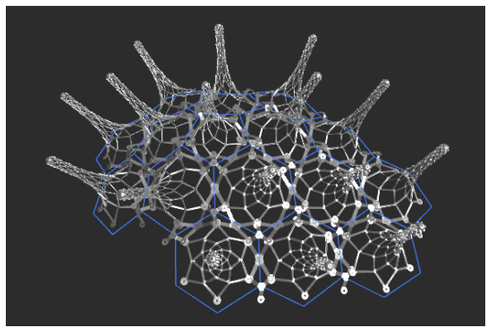
21 For additional information on Living Architecture Polygonal Lattice Design Tool, see the folio linked at LASG.ca.

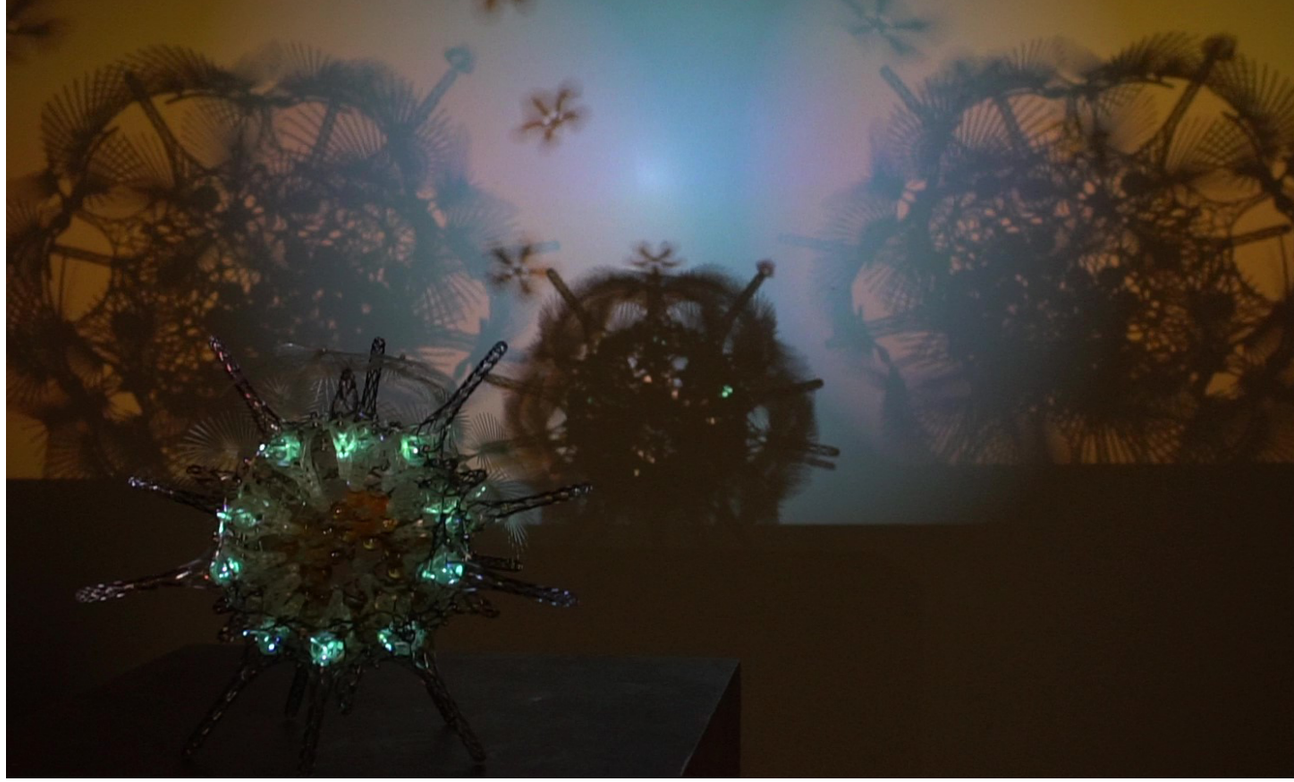
22 Not all components are compatible with all connectors; for example *spars* are compatible with *x-plates,* but not with *tri-plates*. Elements of LASG testbeds that are compatible with each other are considered to be the same "species."

Below Left
A polygonal tiling produced by the LAPL design tool.

Below Right
A polygonal tiling in the LAPL populated with sculpture components from an LASD file.





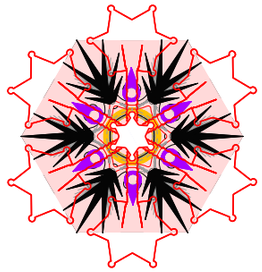


Above
Photograph of *Living Shadows* prototype, Living Architecture Systems Group (2022).

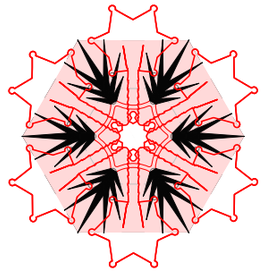
23 For additional information on *Living Shadows*, see the folio linked at LASG.ca.

Augmented Reality Implementation: Living Shadows

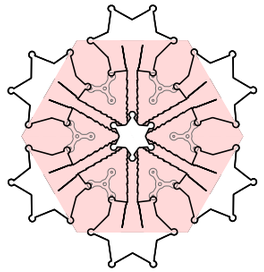
The *Living Shadows* project combines sculpture with animated projection.[23] The projection uses a digital twin of a sculpture object in a game engine environment to augment lighting and shadows cast on the physical object with animated behaviours and shadows of virtual objects. The digital twin, created in the Godot game engine, relies on LASD for import. The Living Shadows importer uses LASD component data to evaluate which sculpture components spawn the creatures that inhabit the virtual world, registering only their shadows in the physical projector-sculpture environment.



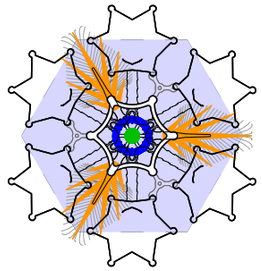
01 F1 - Finial Parent
Sound Sensor Scout



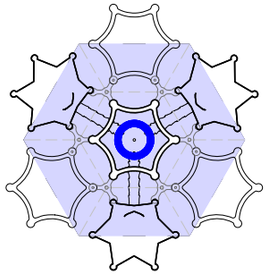
02 F1 - Finial Child
Moth Cluster



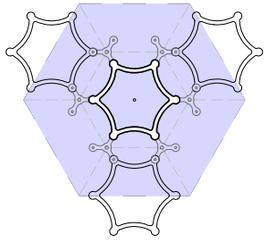
03 F1 - Finial Grandchild
Acrylic Finial Cluster



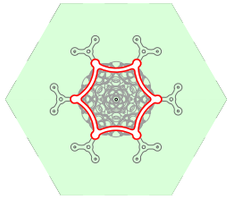
04 P1 - Post Parent
SMA / Breathing Pore



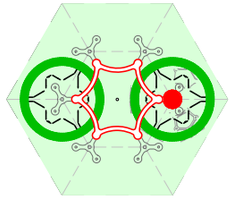
05 P2 - Post Child
Protocell Cluster



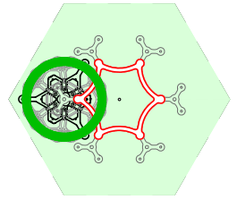
06 P3 - Post Grandchild



07 N1 - Neutral Parent
Hexapod



08 N2a - Neutral Child
Double Nest



09 N2b - Neutral Child
Half Nest

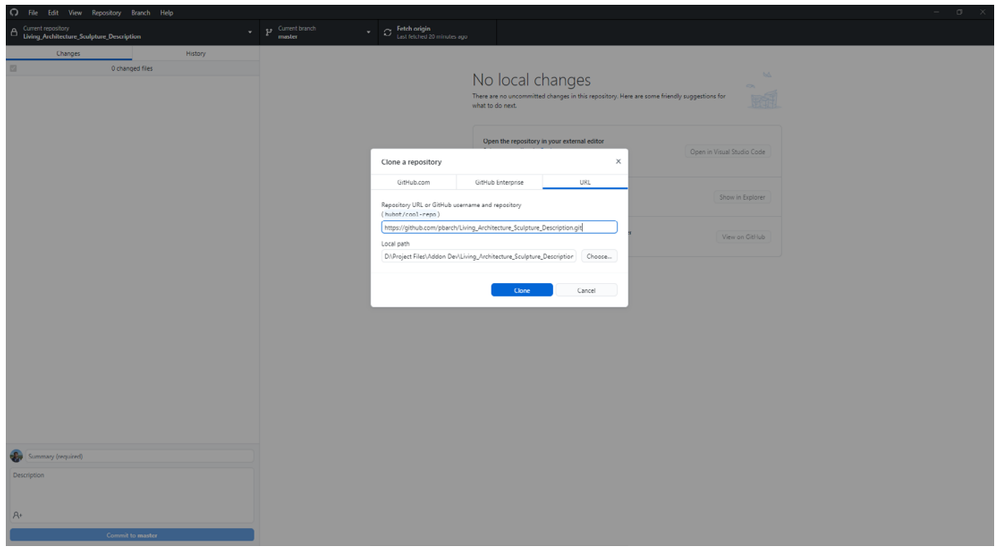
Using the LASD

Facing Page
Working drawings of hierarchy of assembly types included in *Meander's* river.

https://github.com/pbarch/living_architecture_systems_description

Cloning the LASD Repository

The Living Architecture Systems Description and its tools are available online through a central GitHub repository. To access the repository, visit https://github.com/pbarch/Living_Architecture_Systems_Description. The best method to access the repository is to clone it to a local computer. Cloning allows a user's local copy of the software to remain linked to the central development repository, and in turn to be notified when the software is updated, complete with version tracking, release notes, and documentation. GitHub Desktop software provides a user-friendly graphical interface to GitHub's toolkit, and provides an easy way to clone and update software repositories. GitHub Desktop is free to use and can be installed via https://desktop.github.com/.



Above
Cloning the LASD repository using GitHub Desktop.

Once installed, run GitHub Desktop and navigate to the File menu and choose Clone Repository, or use the keyboard shortcut Ctrl + Shift + O on Windows, or ⇧⌘O on macOS. Enter the repository URL and a local path to clone, or download a copy of the repository's contents (https://github.com/pbarch/Living_Architecture_Sculpture_Description.git) from GitHub servers to the local computer.

Once the repository has been cloned, GitHub Desktop offers simple methods for comparing remote changes to local files, and for syncing those files when desired. GitHub Desktop's Fetch Origin and Pull Origin commands handle comparing and syncing, respectively.

From the local repository, users can install the relevant LASD Toolkits to desired software.

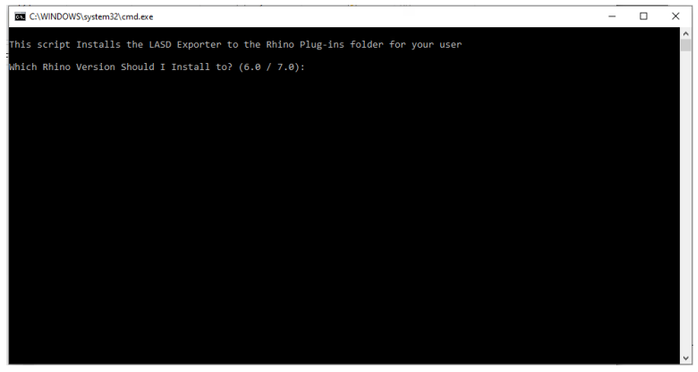
Using LASD with Rhino

This section is a guide to installing the LASD Toolkit plugin to work with Rhino 6 or 7 on Windows, and to the LASD Toolkit Rhino commands included with this plugin.

1. Run the install script

To install the LASD toolkit for Rhino, navigate to the directory of the local repository. Within this folder, the `LASD_Rhino_Install.bat` file is located at:

```
..\RhinoScript\LASD_Rhino_Install.bat
```



Right
The LASD Rhino Installer.

From Windows Explorer, right-click `LASD_Rhino_Install.bat` and select "Run as Administrator". This will raise a Command Prompt, where users are prompted to enter their Rhino version. After keying in the version number and pressing enter, the batch script proceeds to install the included scripts to the appropriate Rhino plugin folders. The installer creates a symbolic link, or symlink, within Rhino's plugin folder that links to the local repository. This allows users to keep the Rhino plugin up to date simply by fetching changes from the remote (GitHub) repository, and eliminates the need to run the installer each time the repository is changed.

2. Start Python in Rhino

Before using the LASD Toolkit commands, run `EditPythonScript` in Rhino to ensure Rhino has initialized the Python backend that supports these commands. Close the Rhino Python Editor window that appears–Python will continue to be available in the background of the active Rhino session.

3. LASD Commands

At the time of writing, the following Rhino commands are provided by the LASD Toolkit:

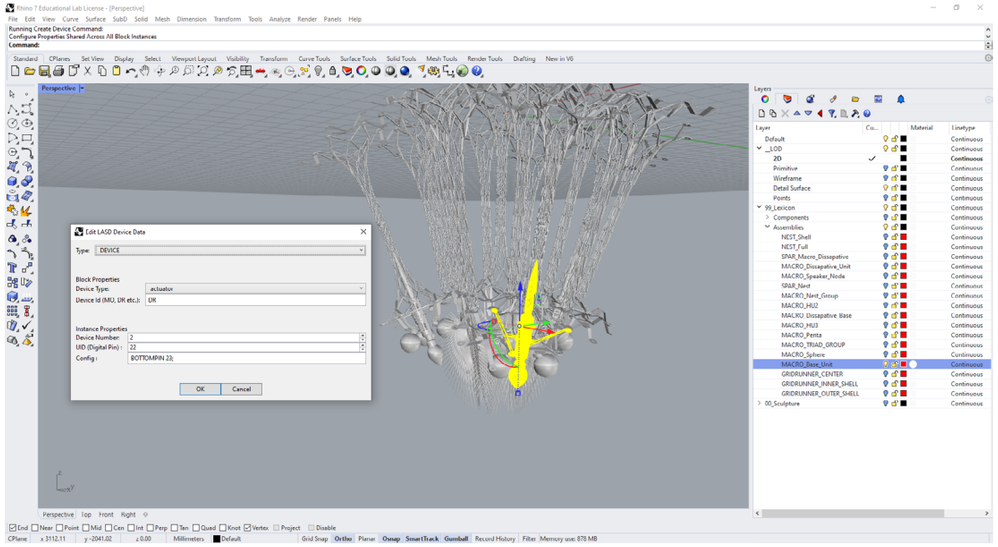
`ImportLASD`: reads an LASD .json file from a specified file and generates block definitions and instances accordingly, with options to reference a Lexicon Mesh Library and to generate primitive geometry according to LASD specification.

`ExportLASD`: writes an LASD .json file.

`CreateLayersLASD`: creates a standard layer structure, complete with sculpture, lexicon and level of detail top-level layers.

`BlockLASD`: provides a wrapper for Rhino's normal Block command to easily define a block for use with LASD. When a block is defined using BlockLASD, an instance is also placed on a layer in the lexicon section, named according to the name of the block.

`DeviceLASD`: opens a dialog to add or edit device data attached to blocks that represents sensors, actuators, and control devices.



Above
The DeviceLASD dialog in Rhino.

Optional: Rhino Plugins Directory

If you need to delete the LASD toolkit from Rhino, the plugin installation directories are as follows:

For Rhino 7:

Windows:
`%APPDATA%\McNeel\Rhinoceros\7.0\Plug-ins\PythonPlugins\`

macOS:
`~/Application Support/McNeel/Rhinoceros/7.0/Plug-ins/PythonPlugIns/`

For Rhino 6:

Windows:
`%APPDATA%\McNeel\Rhinoceros\6.0\Plug-ins\`

macOS:
`~/Application Support/McNeel/Rhinoceros/6.0/Plug-ins/`

Using LASD with Blender

This section is a guide to installing and using the LASD Toolkit add-on to work with Blender on Windows.

1. Run the install script

To install the LASD toolkit for Rhino, navigate to the directory of the local repository. Within this folder, the `LASD_Blender_Install.bat` file is located at:

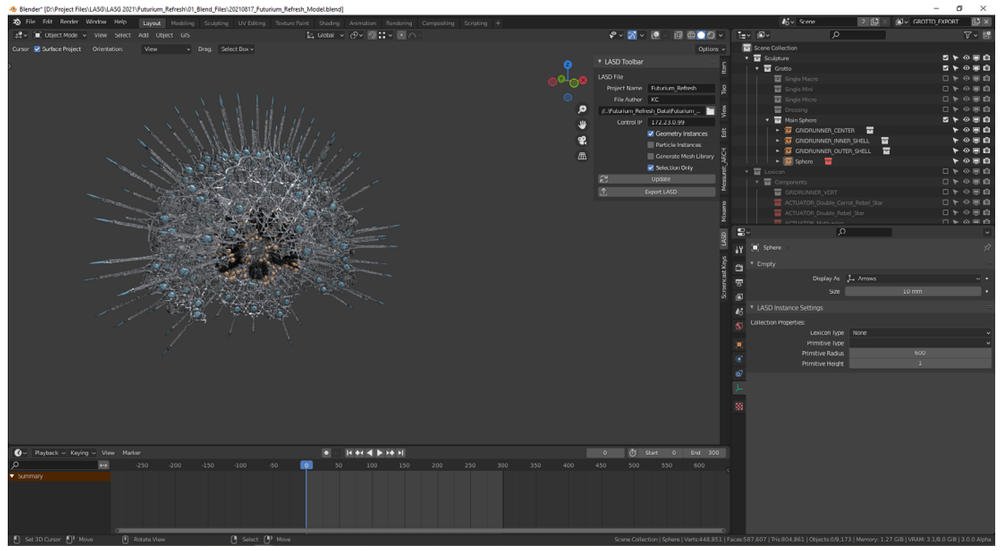
```
..\BlenderAddon\LASD_Blender_Install.bat
```

From Windows Explorer, right-click `LASD_Blender_Install.bat` and select “Run as Administrator”. This raises a Command Prompt, where users are prompted to enter their Blender version. After keying in the version number and pressing enter, the batch script proceeds to install the included scripts to the appropriate Blender Add-On folders. The installer creates a symbolic link, or symlink, within Blender’s “addon” folder that links to the local repository. This allows users to keep the Blender Add-On up to date simply by fetching changes from the remote (GitHub) repository, and eliminates the need to run the installer each time the repository is changed.

2. Activate the add-on

Open Blender and go to the User Preferences (Edit > Preferences). Select the Add-ons tab and search for the Living Architecture Sculpture Description add-on. Click the checkbox to enable it.

With the add-on enabled you should now have an “LASD” tab in the 3D view toolbar, as well as LASD-specific instance and collection properties available in the properties menu.



Above
The LASD user interface in Blender.

Optional: Blender Add-ons Directory

If you need to delete the LASD toolkit from Blender, the the add-on installation directories are as follows:

Windows:

```
%APPDATA%\Blender Foundation\Blender\
<version-number>\scripts\addons\
```

macOS:

```
~/Library/Application Support/Blender/
<version-number>/scripts/addons/
```

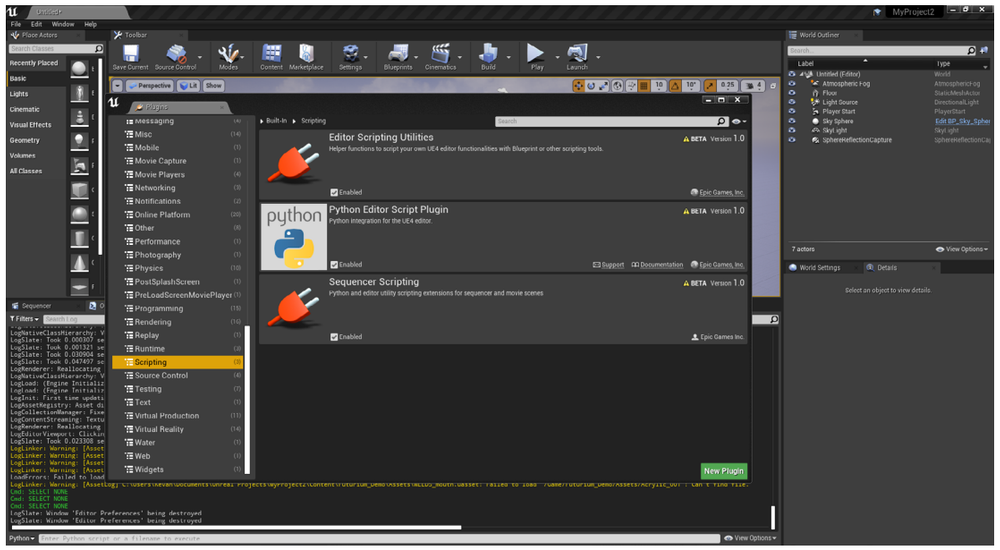
Using LASD with Blender

This section is a guide to using the LASD import script in an Unreal Engine project.

1. Enable Python scripting in Unreal Engine

To enable Python scripts to run in Unreal Engine, select Edit > Plugins, and under the Builtin section of the browser, select Scripting. Ensure that Editor Scripting Utilities, Python Editor Script Plugin, and Sequencer Scripting are all enabled. Once all three are enabled, Unreal will prompt you to restart the engine.

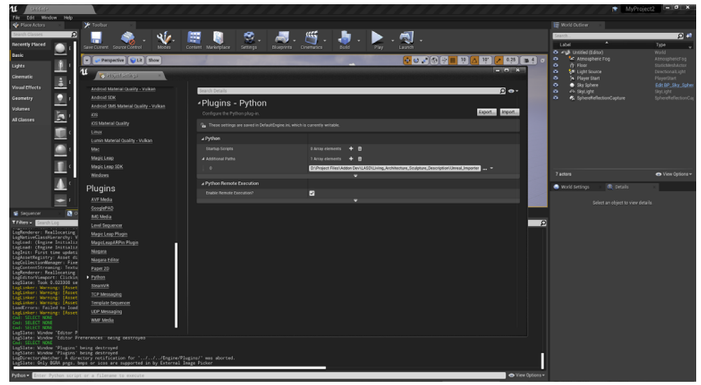
Below
Enabling Python scripting in Unreal Engine.



2. Provide Unreal the path to the LASD import script

Go to Edit > Project Settings, and under the Plugins section of the browser select Python. Next to the Additional Paths field click the plus button to add a path to a Python script. In the text box that appears enter the path to the Unreal_Importer folder in the LASD Repository. Once you've entered the path, Unreal will prompt you to restart the engine.

Right
Setting the import script path in Unreal Engine.



3. Enable the Output Log

Under Window > Developer Tools, enable the Output Log. Select the Output Log (it should appear as a tab next to the Content Browser). WIth the Output Log open, change the log's mode from Cmd to Python by selecting "Python" in the menu at the bottom left of the output log

4. Load and run the import script

In the text box of the Output Log, enter the following commands:

```
import UnrealLASDImport as LASD
```

This will load the Python module defined by the import script.

```
LASD.importLASD(r "\\your\\path\\to\\an\\LASD.json" )
```

This command runs the importLASD function to import the LASD file from the path provided. The import function can be executed with a number of optional arguments, which can be added, comma-separated, after the filepath.

The optional argments are:

`import_only = False,` a True or False value, if True, the import function will only import the assets from the Lexicon Mesh Library, and will not populate the scene

`populate_only = False,` a True or False value, if True, the import function will only populate the scene from already imported assets in the Unreal project.

`uv_only = False,` a True or False value, if True, the import function will only UV unwrap assets already in the Unreal project

`num_batches = 5,` an Integer value, splits the scene population process into a number of batches. This reduces RAM usage, which can be quite high during scene population.

`batch = "all" ,` can be either a range "[0,2]" or "all". If a range is specified, only those batches will be imported when running the script. This further reduces RAM usage.

Note: The arguments shown are optional. If excluded, these arguments default to the values listed here. The import function will run these operations in sequence.

Using LASD with Testbed-Control

To use an LASD file with Testbed-Control, the following keys need to be added to the options.json configuration file:

`"file_name" : "Futurium_Refresh" ,` Make sure the file name specified here matches the filename of your LASD .json file.

`"use_lasd_devices" : "true" ,` A True or False value. Setting this to True will load device data from the "Testbed-Control\\Simulator\\LASD_Data" folder. Setting this to False will fall back to loading device data from a device locator.csv file

`"use_lasd_gridrunner" : "true" ,` A True or False value. Setting this to True will create gridrunner verts from the optional gridrunner key in an LASD file. Setting this to False will fall back to creating gridrunner verts from a separate gridrunner.json file

`"use_lasd_model" : "true" ,` A True or False value. Setting this value to True will use the primitive data on the LASD Lexicon elements to construct the low poly sculpture model from the LASD data. Setting this to False will fall back to loading the sculpture model from an .obj file.

Additionally, an optional key can be provided:

`"LASD_data_path" : "\\your\\path\\to\\an\\LASD.json"`

This will override the default behaviour of loading the LASD file located at `"Testbed-Control\\Simulator\\LASD_Data"` and instead load the LASD file from the specified path. This can useful when testing and iterating on files, as this path can be set to your DCC's LASD export path, allowing the Testbed-Control data to update to newly exported data automatically on restart, without needing to copy the exported LASD file into the `"Testbed-Control\\Simulator\\LASD_Data"` folder.

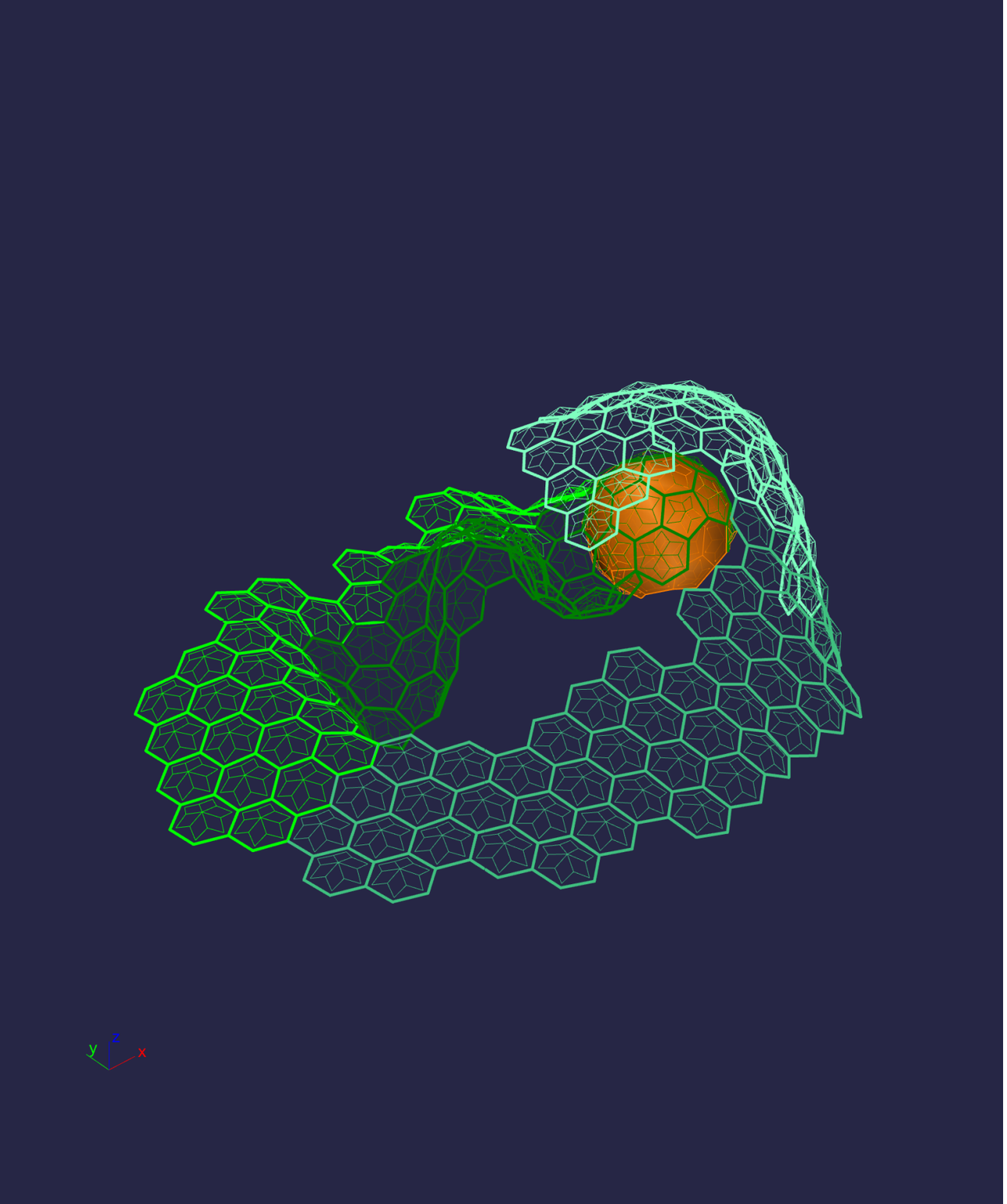
```python
def import_function():
  # The main function would be called by the importer (i.e. "RunCommand" in the Rhino Importer).

  lasd_dictionary = get_lasd_file()

  instances = lasd_dictionary['instances']
  components = lasd_dictionary['lexicon']['components']
  assemblies = lasd_dictionary['lexicon']['assemblies']

  for component in components:
    generate_lexicon_component(component)

  for assembly in assemblies:
    generate_lexicon_assembly(assembly)

  generate_instances(instances)


def get_lasd_file() -> Dictionary:
  # This function prompts the user for the LASD folder path and parses the .json file to a dictionary.

  return lasd_dictionary

def generate_lexicon_component(component) -> None:
  # Creates a software specific definition of the component (i.e. block) located at the Origin
  # to be instanced later. Optionally imports mesh representations from the Lexicon Mesh Library.
  # Handles other software specific tasks for the component, layer assignment, LOD generation etc.

  component_definition = make_definition(containing = [], name = component['name'], position = (0,0,0))

def generate_instances(instances) -> List:
  # This function takes either the list of nested instances from an assembly or the list of sculpture
  # instances. Be sure to convert to the coordinate space of your target software, as nested instances
  # from assemblies may refer to other assemblies before they have been created. Be sure to check if an
  # assembly exists before trying to instance it.

  created_instances = []
  for instance in instances:
    instance_definition = instance['instance']
    transforms = read_and_convert_transforms()

    if instance_definition.exists_in_software_definitions():
      new_instance = create_instance(instance_definition, transforms)
      created_instances.append(new_instance)

    else:
      generate_lexicon_assembly(lexicon['assemblies']['instance_definition'])

  return created_instances

def generate_lexicon_assembly(assembly):
  # This function creates a software specific definition of an assembly (block etc.), containing other
  # components or assemblies located at the origin to be instanced later

  if assembly_definition.exists_in_software_definitions():
    print('Already Exists')
    return

  nested_instances = assembly['nested_instances']
  assembly_instances = generate_instances(nested_instances)
  make_definition(containing= assembly_instances, name = component['name'], position = (0,0,0))
```


Writing an Importer for the Living Architecture Systems Description

Facing Page
Pseudo-code for a Python LASD importer.

While the specifics of creating an importer will vary based on the available API of the target software package, the core steps involved would be largely the same. To help guide the creation of new import plugins, pseudo code for a bare bones LASD importer is provided here at https://github.com/pbarch/Living_Architecture_Systems_Description/blob/master/Pseudo_Importer.py.

For a more complete reference of an actual implementation, see the Rhino Importer at https://github.com/pbarch/Living_Architecture_Systems_Description/blob/master/RhinoScript/LASDManager%20%7Bfd72a2df-89d1-42e5-a465-9059768362ce%7D/dev/ImportLASD_cmd.py.



Next Steps for Development

Facing Page
Screenshot showing a polygonal wireframe working model in Rhino. Models are constructed procedurally around precise topological relationships between components and groups within a sculpture. Formative meta-geometries are the basis of models like this, but are not systematically recorded or transmitted within the LASD schema.

The LASD provides a framework to describe the hierarchy of assemblies and components that make up LASG testbeds, their physical layout, and the topology of the network of sensors, actuators, and microprocessors that bring the testbeds to life. It facilitates the exchange of this data between a variety of digital design tools and allows for streamlined workflows for rendering, design, simulation, and control of living architecture environments.

However, the foundational underlying meta-geometries that structure the composition of components and assemblies, and the parametric systems that drive their form, are not currently transmitted using or documented within the LASD. The current version of LASD supports recording and transmitting meta-geometry definitions by using custom component data, as discussed in the *Futurium Noosphere* case study. A next step in LASD development would be the creation of a formal system that describes foundational structures. A future version of the LASD may also include behaviour settings that dictate how elements of a testbed respond to different conditions and stimuli, which are also currently stored using JSON formats. As the complexity of responsive architectural environments increases, the LASD will need to evolve to meet the needs of designers.

References

Gilbert, Scott F. and Sahotra Sarkar. “Embracing Complexity: Organicism for the 21st Century.” Developmental Dynamics: An Official Publication of the American Association of Anatomists 219, no. 1 (2000): 1–9.

“GITF/Readme.md at Main · KhronosGroup/GLTF: Design Goals.” KhronosGroup. GitHub. Accessed January 14, 2023, https://github.com/KhronosGroup/glTF/blob/ main/specification/1.0/README.md#designgoals.

“GITF/Readme.md at Main · KhronosGroup/GLTF: Introduction.” KhronosGroup. GitHub. Accessed January 14, 2023, https://github.com/KhronosGroup/glTF/blob/ main/specification/1.0/README.md#introduction.

“GITF/Readme.md at Main · KhronosGroup/GLTF: Motivation.” KhronosGroup. GitHub. Accessed January 14, 2023, https://github.com/KhronosGroup/glTF/blob/ main/specification/1.0/README.md#motivation.

“GITF - Runtime 3D Asset Delivery.” The Khronos Group. Accessed December 3, 2020, https://www. khronos.org/api/index_2017/gltf.

“Introducing Json.” JSON. Accessed January 13, 2023. https://www.json.org/json-en.html.

“Introduction to USD — Universal Scene Description 22.11 Documentation.” Pixar Animation Studios. Accessed December 8, 2022, https://graphics.pixar.com/usd/release/intro.html.

“Introduction to USD: Why Use USD?.” Introduction to USD - Universal Scene Description 22.11 documentation. Accessed January 14, 2023, https://graphics.pixar.com/usd/release/intro.html#whyuse-usd.

Merriam-Webster.com Dictionary, s.v. “Object-oriented programming.” Accessed January 13, 2023, https://www.merriamwebster.com/dictionary/objectoriented%20programming.

“Rhino.Python Guides.” Rhino Developer, Robert McNeel & Associates. Accessed January 19, 2023, https://developer.rhino3d.com/guides/rhinopython/.

Youngs, Amy M. “The Fine Art of Creating Life.” Leonardo 33, no. 5 (2000): 377–80.