SD2858

Extending Autodesk Forma to Create Powerful Analysis Solutions for the AEC Industry

Håvard Høiby
Autodesk

Kean Walmsley
Autodesk

Learning Objectives

Learn about the benefits of using Autodesk Forma for early-stage planning and design.

Learn the various ways to extend Autodesk Forma, including custom views, data providers, and HTTP APIs.

Learn how you can integrate WebAssembly packages to implement complex, real-time analysis routines.

Description

Autodesk Forma software is Autodesk’s cloud-based solution for early-stage planning and design. Aside from having a rich, evolving feature set, Autodesk Forma is also a platform: It exposes a powerful set of APIs that allow you to extend the core functionality with custom data and analyses. This session will look at the various ways to extend Autodesk Forma, with a particular focus on how Autodesk Research has created a voxel-based analysis extension using the embedded views software development kit (SDK).

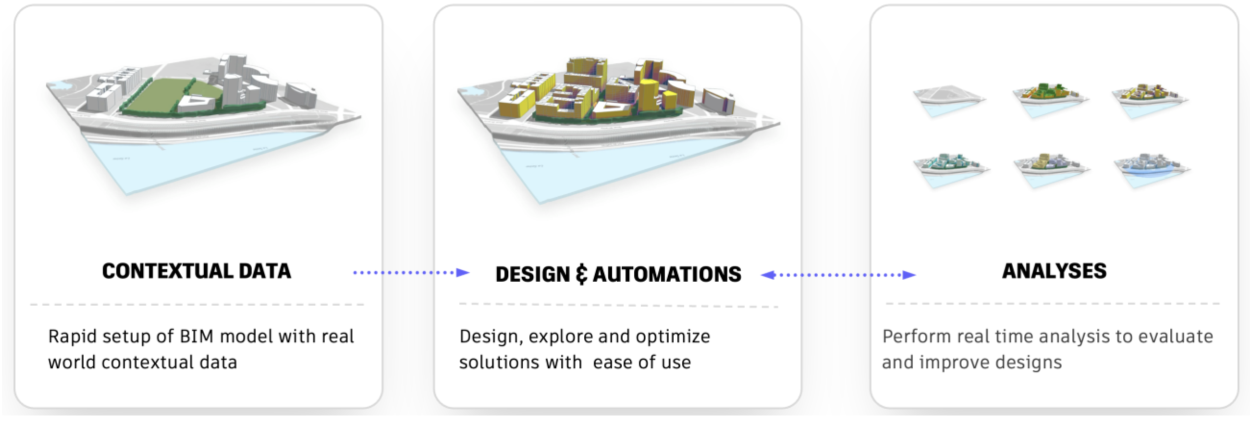
Speakers

Håvard is a Principal Software Engineer at the Autodesk Forma team in Oslo, Norway. He joined Spacemaker AI in 2018 and followed the company through the scale-up phase and the acquisition by Autodesk. For Spacemaker and now Forma he has built ray-tracing analyses like daylight and view analyses, worked on the contextual data pipeline and most recently joined the efforts to open up APIs and to connected to Forma to the 3rd party ecosystem inside and outside of Autodesk.

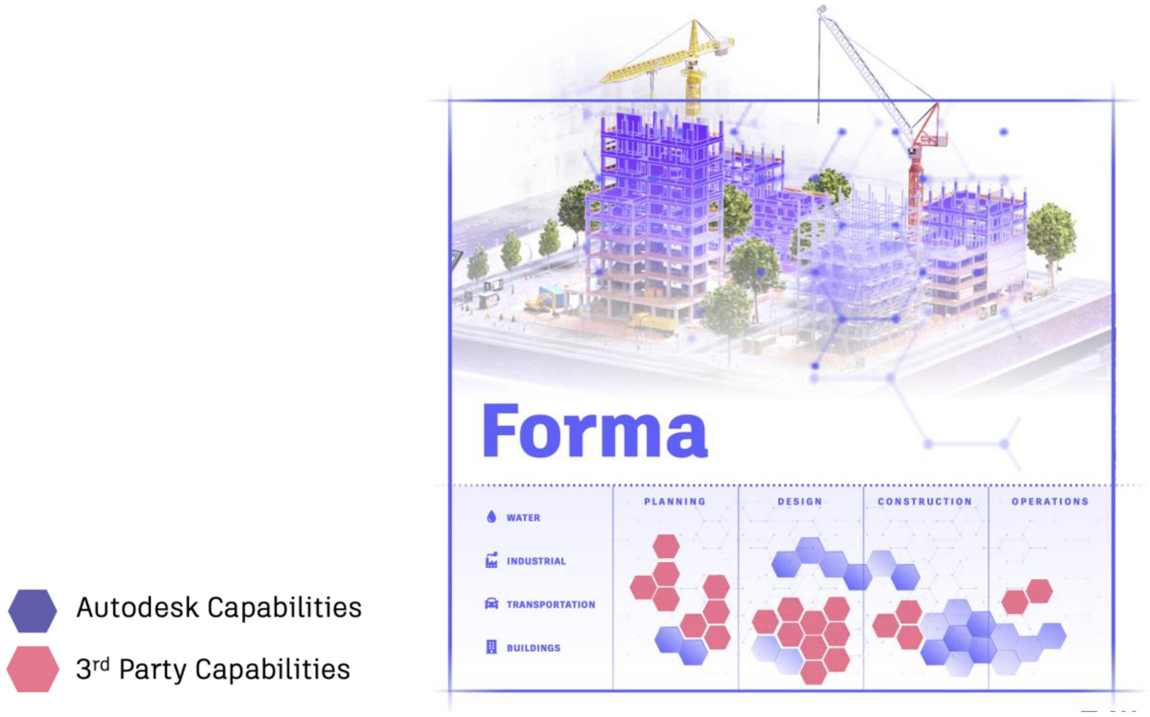
Kean is a Director of Systems Design/Architecture Engineering at Autodesk Research and is primarily focused on the research area of human-centric building design. He has previously worked on projects exploring the integration of IoT data with BIM (Digital Twins) using Autodesk Platform Services, as well as Generative Design in the AEC space. He has worked in various roles – and in various countries – during his career at Autodesk, including building and managing teams of software developers in Europe, the Americas, and Asia/Pacific. Kean engages regularly with Autodesk’s developer and computational design communities, providing technical content and insights into technology evolution.

Autodesk Forma

The Autodesk Forma In-Market offering provides a suite of tools for early stage building design. This is the first iteration on our Journey towards an Industry Cloud for the AECO industry. Forma consists of 3rd party data providers, design and automations tools for 3d conceptual massing and a large suite of environmental analyses.


CONTEXTUAL DATA
Rapid setup of BIM model with real world contextual data
DESIGN & AUTOMATIONS
Design, explore and optimize solutions with ease of use
ANALYSES
Perform real time analysis to evaluate and improve designs

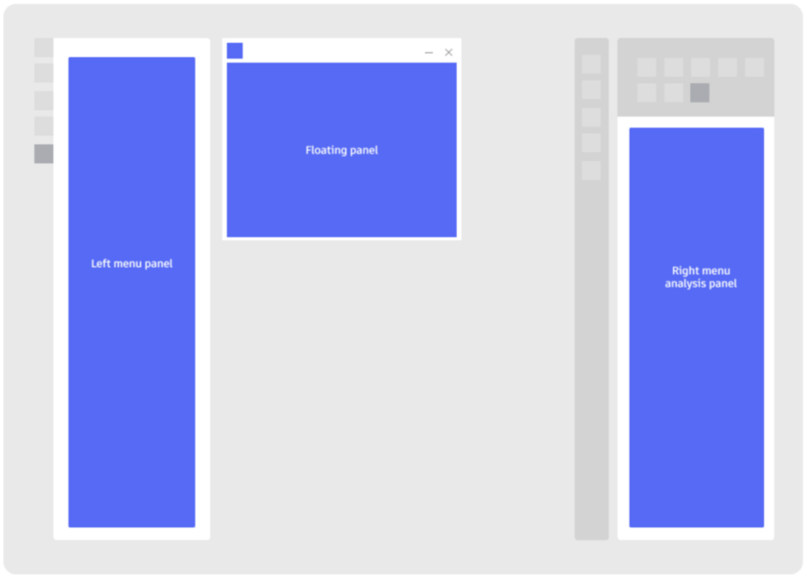
With Forma we at Autodesk are building a cloud based platform for the future. We aim to unify workflows across personas in an open way. An important part of this journey is to enable 3rd parties to build upon and extend Forma.



The Forma Industry Cloud as built by Autodesk and 3rd parties.

Forma Extensions

Forma allows 3rd party developers to extend Forma through a SDK and Embedded Views. With these two technologies 3rd parties can add their own UI to Forma and interact with the 3d scene and the Forma data model.



Embedded Views

A simple extension

The simplest Forma Extension is a html page with a script tag including our SDK.

```
<html>
  <body>
    <script type="module">
      import { Forma } from "https://esm.sh/forma-embedded-view-sdk@0.80.0/auto";

      console.log(Forma.getProjectId());
    </script>
  </body>
```

Hosting this on a public URL and registering this as an extension through the Extension management panel in Forma is all that is needed to get started.

To add UI to a Forma extension you simple write standard html, css and Javascript inside this web page. There are some design guildelines you need to comply to, and we host a component library.

Read more about this in our documentation:
https://aps.autodesk.com/en/docs/forma/v1/overview/welcome-to-forma/

Using the Forma SDK

The Forma SDK is distributed as the forma-embedded-view-sdk on npmjs.com. You can install it using Javascript tooling such as yarn and npm.

With it you can:

- Control the 3D Scene
- Read Forma elements
- Write Forma elements
- Render to the 3D scene
- Run analyses and consume their results.

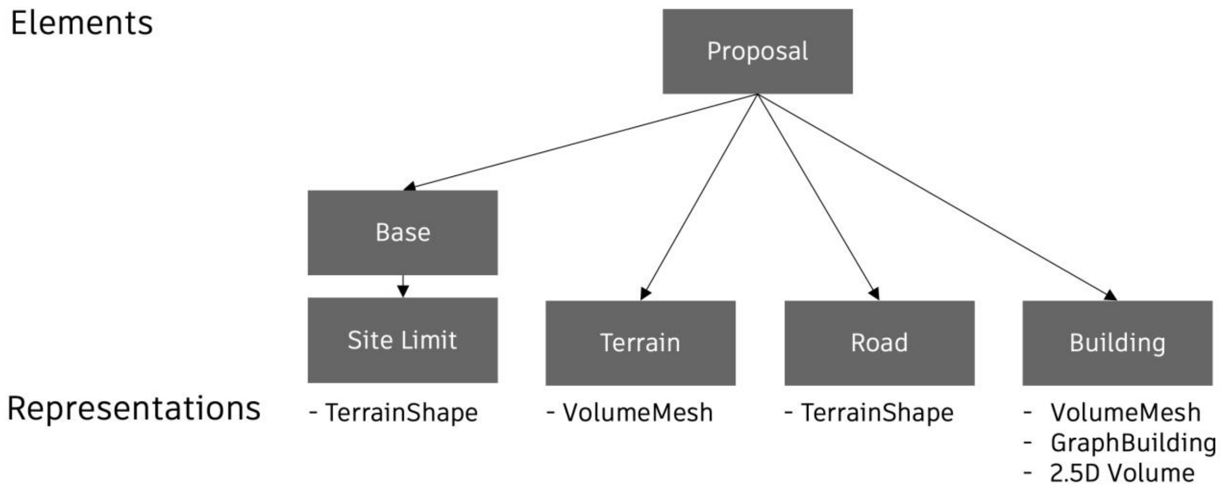
Refer to the full SDK API reference here: https://app.autodeskforma.com/forma-embedded-view-sdk/docs/index.html

Control the 3D Scene

With the Forma.sun package you can control the position of the simulated sun and shadow effects in the Forma 3d scene. The Forma.camera package lets you control the position of the camera in the scene. Our Forma.designTool package allows the extension author to request the user to draw a point, line, polygon or extruded polygon with the built in drawing tools in Forma. The resulting geometry is returned to the extension code. Lastly our Forma.colorBar api lets the extension add a visual color bar to the Forma UI to explain any colors used by the extension in the 3d scene.

Reading Forma Elements

From a consumer side the Forma data model is built up of two pieces. Elements with their relationships and Properties, and Representations. Elements are anything in the scene or internal organization of the elements in the scene. Representations are the way a Forma element communicates its internal state like a Volume Mesh to the rest of Forma. The element keeps its internal state, called its Design Intent, private.



An abstract view of the Forma data model

An element is a JSON document, you can get it by calling Forma.elements.getByPath.

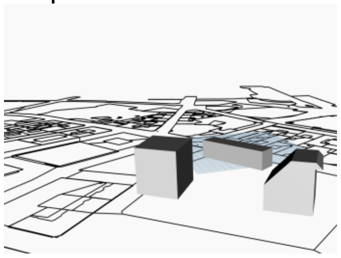
```
const { element } = await Forma.elements.getByPath({ path: "root" });
```

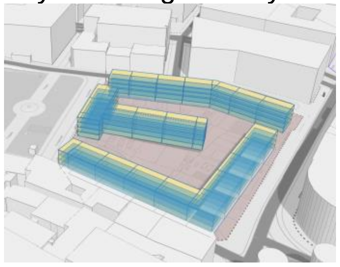
This will return the element as a Javascript object:

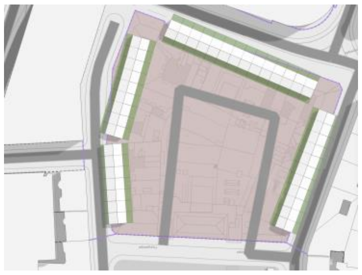
```
{
  urn: "urn:adsk-forma-elements:proposal:pro_mnpo1vppxk:...:..",
  properties": {category": "proposal"
  },
  children: [{
    "key": "d50cd6f",
    "urn": "urn:adsk-forma-elements:basic:pro_mnpo1vppxk:...:...",
  ],
  representations: {
    volumeMesh: {
      type: "linked",
      blobId: "integrate:<>",
      selection: { "value": "95e72063e1b03", "type": "equals" }
    }
  },
}
```

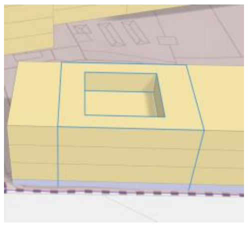
Read Representation

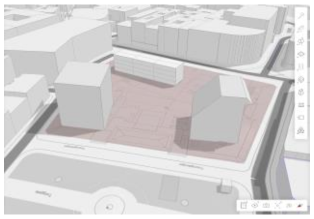
The representations in Forma is how you read geometry from the elements.

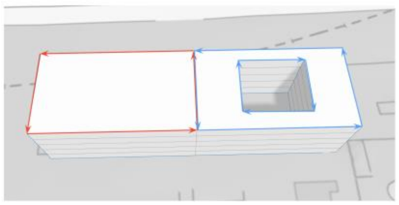

Footprint


Gross Floor Area


Terrain Shape


Volume 2.5D Collection


Volume Mesh


Graph Building

There is a simple way to request some of these representations:

```
Forma.geometry.getTriangles({ path });
```

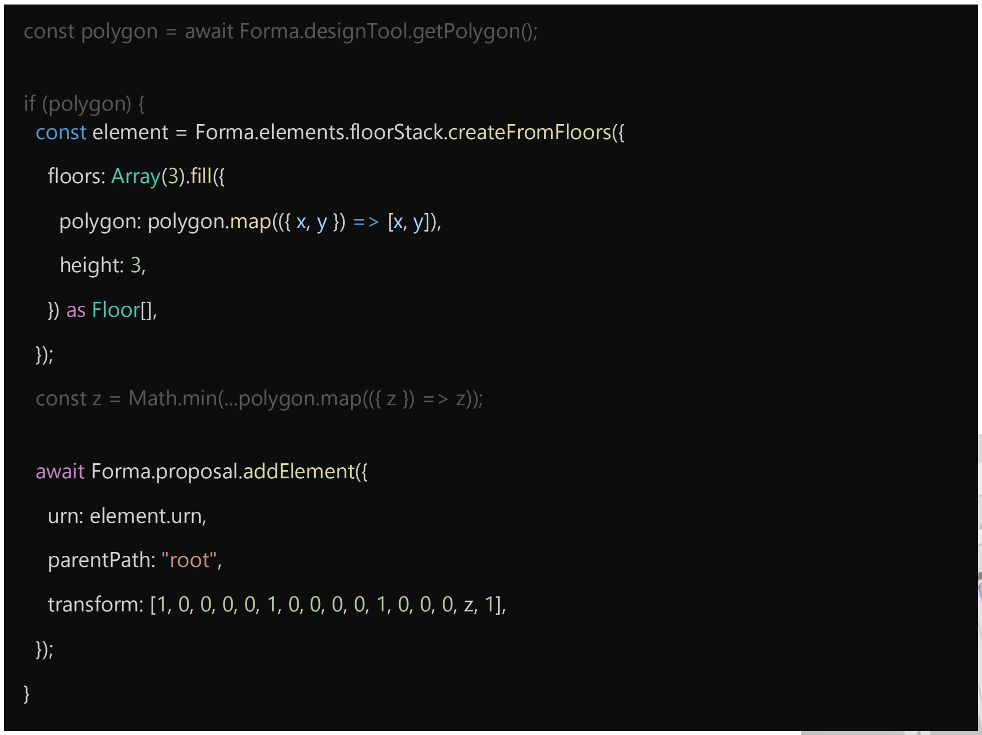
And there is a nuts and bolts way:

```
const { element } = await Forma.elements.getByPath({ path });
const volumeMesh = element.representations?.volumeMesh;
if (volumeMesh) {
  const representation = Forma.elements.representations.volumeMesh({
    urn: element.urn
  });

  // Select glb node based on selection criteria
  const selectedVolume = getSelected(
    representation,
    volumeMesh.selection
  );
  const transform = Forma.elements.getWorldTransform({ path });

  // Apply affine transform on glb
  const volume = transform(selectedVolume, transform);
}
```

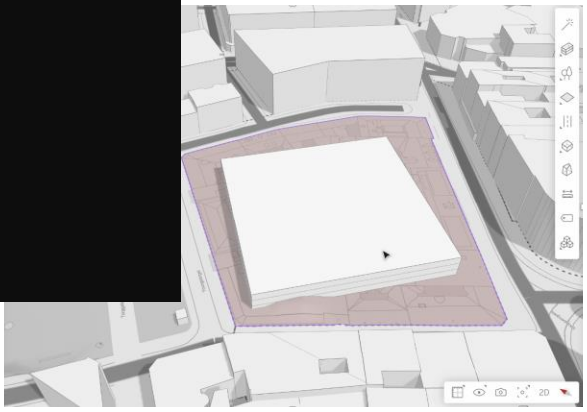
Write Elements

You can add geometry as elements into the Forma data model. We expose a generalized Forma.integrateElement which lets to create any element with any representation. There are some simple ways for adding GeoJSON and GLB elements. We also have a few specialized ways, like the Forma.elements.floorStack for creating buildings, Forma.proposal.replaceTerrain for replacing the terrain element or Forma.geoData for adding GeoJSON with given in a known projection as buildings, roads or property boundaries.

After creating these elements you can use the Forma.proposal.addElement method to add the created element to the current open proposal.



```
const polygon = await Forma.designTool.getPolygon();


if (polygon) {
  const element = Forma.elements.floorStack.createFromFloors({
    floors: Array(3).fill({
      polygon: polygon.map(({ x, y }) => [x, y]),
      height: 3,
    }) as Floor[],
  });
  const z = Math.min(...polygon.map(({ z }) => z));

  await Forma.proposal.addElement({
    urn: element.urn,
    parentPath: "root",
    transform: [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, z, 1],
  });
}
```

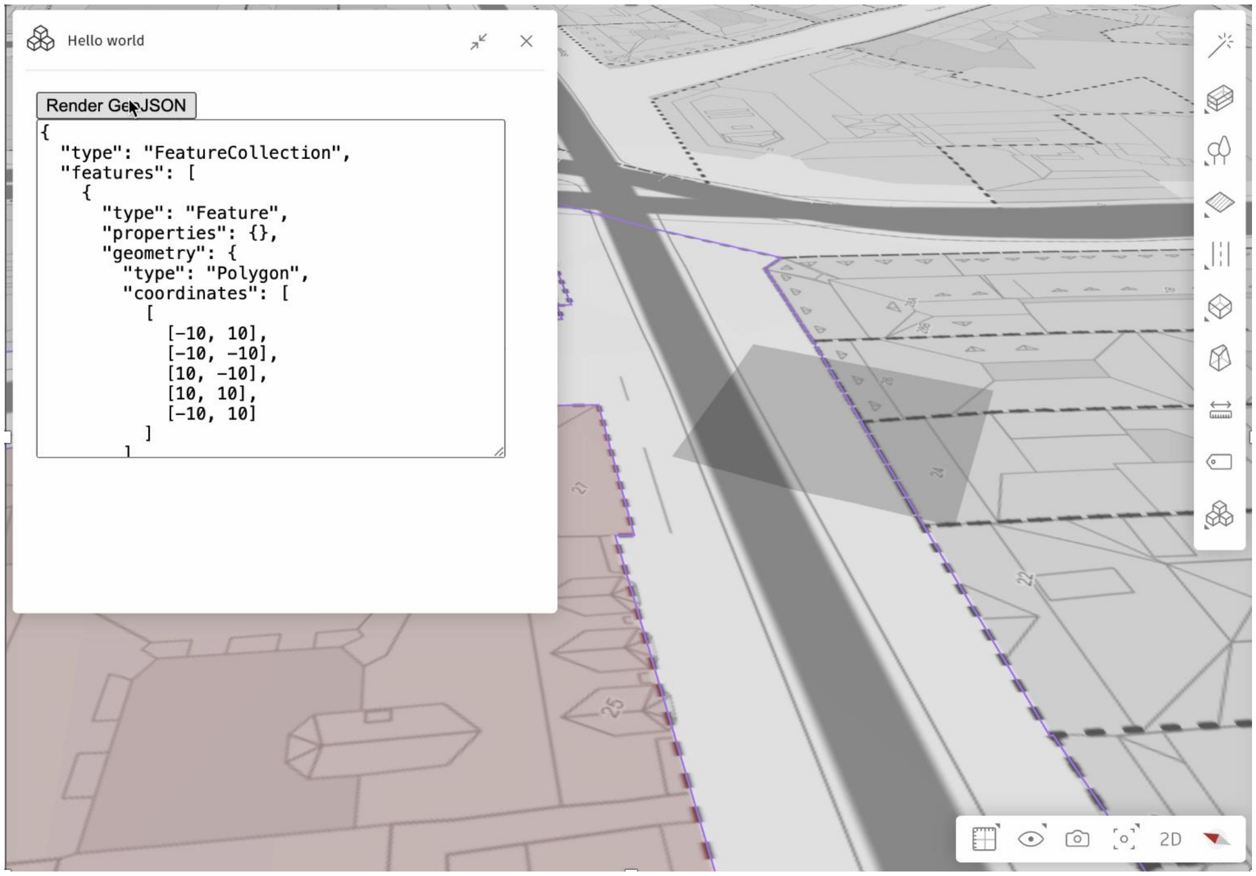


You can also write properties to existing elements using Forma.elements.editProperties and applying the Forma.properties.replaceElement method to update an element with a new version.

Render to the 3d scene

For some use cases you want to add visualization which should not persist into the data model of a proposal. For these you can use the Render package under Forma.render. With this package we aim to expose methods close to widely known and used formats. With Forma.render.glb you can render 3d geometry in a glb file, which is the binary version of GTLF. With Forma.render.geojson you can render GeoJSON onto to terrain element. With Forma.terrain.groundTexture you can render a HTMLCanvas onto the terrain element, and lastly if you've only got triangles you can use Forma.render.addMesh to render index or unindex vertices with colors and normal to the 3d scene.

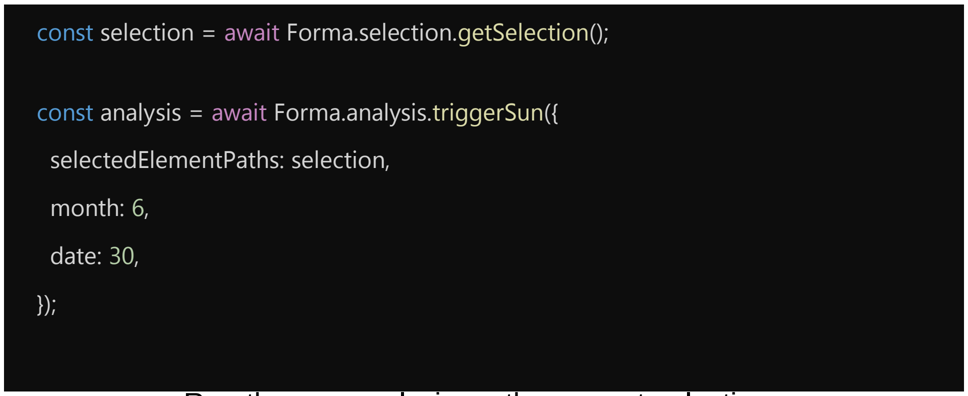
```
const { id } = await Forma.render.geojson.add({ geojson });
```



```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [-10, 10],
            [-10, -10],
            [10, -10],
            [10, 10],
            [-10, 10]
          ]
        ]
```

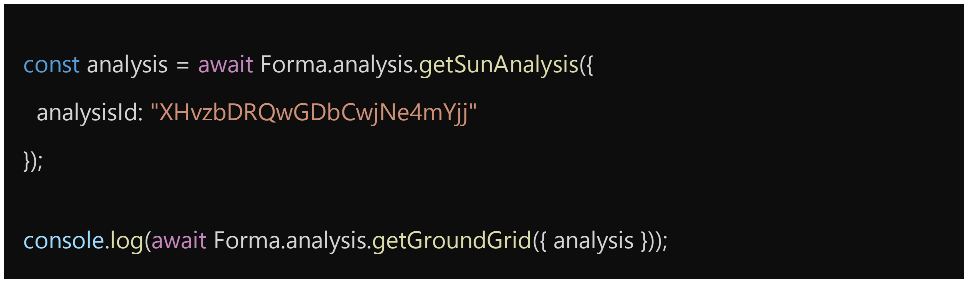
Controlling Forma Analyses

With the Forma.analysis package you can run and consume the built in Sun and Noise environmental analysis. With Forma.predictiveAnalysis.predictWind you are able to use the wind surrogate model to predict the wind conditions.



```
const selection = await Forma.selection.getSelection();

const analysis = await Forma.analysis.triggerSun({
  selectedElementPaths: selection,
  month: 6,
  date: 30,
});
```

Run the sun analysis on the current selection



```
const analysis = await Forma.analysis.getSunAnalysis({
  analysisId: "XHvzbDRQwGDbCwjNe4mYjj"
});

console.log(await Forma.analysis.getGroundGrid({ analysis }));
```

Consume the ground grid for a given sun analysis.

Refer to our documentation for a description of the analysis results:
https://aps.autodesk.com/en/docs/forma/v1/embedded-views/useful-concepts/analysis/

Resources and HTTP API

Some useful samples and references

Docs: https://aps.autodesk.com/en/docs/forma/v1/reference/
Dynamo Extension: https://github.com/spacemakerai/forma-dynamo-extension
Shadow Study: https://github.com/autodesk-platform-services/aps-forma-extension-shadow-study
Extension Samples: https://github.com/spacemakerai/forma-extensions-samples
API Samples: https://github.com/spacemakerai/public-api-examples

HTTP API

This talk and handout focuses on extending Forma through Embedded Views and the forma SDK. Most of the functionality, expect for those controlling the 3d scene, is exposed through the SDK is also available through our public HTTP API.

- Library – Interact with the Forma Element Library
- Project – Read project data

- Elements – Read the Forma element model

- Proposal – Write proposal elements
- Geometries – Write basic elements
- Integrate – Write Integrate elements
- Terrain – Write Terrain elements

Refer to our API reference for more details:
https://aps.autodesk.com/en/docs/forma/v1/reference/http-reference/

Autodesk's Research into Human-Centricity of Buildings

This part of the session is focused on a technology that has been developed by Autodesk Research to allow exploration of 3D space by breaking it down into voxels. The term voxel refers to the idea of a "volumetric pixel": while pixels exist in only two dimensions, voxels exist in three.

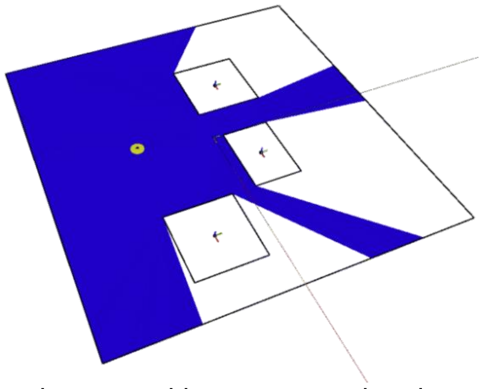
Why voxels?

Voxels have been popularized by the Minecraft video game and it's a great way to imagine the idea of a voxel.

A Minecraft model of the Samuel-Novarro House, created by Kevin Shull and shown on YouTube.

The fact that Minecraft has been used to depict architectural spaces such as the Samuel-Novarro house designed by Lloyd Wright is actually not especially relevant here (although it is very interesting in and of itself). We're more interested in how this representation might be used to understand space.

Most existing approaches for understanding the composition of space revolve around the use of vectors rather than voxels. For instance we can use a technique called raycasting (commonly used in computer graphics) to create an isovist of the visibility from a point in a space.



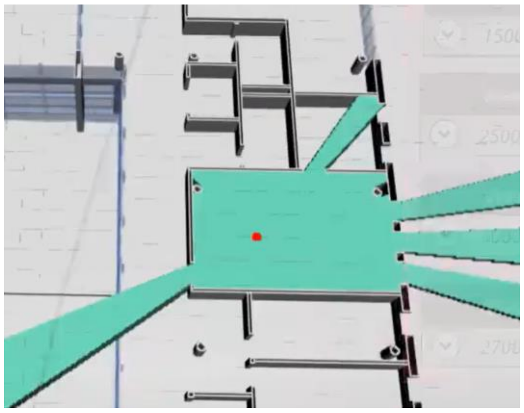
A vector-based isovist created in Dynamo using the Refinery Toolkit

This works well but can be time-consuming and computationally expensive, especially when handling repetitive tasks.

One of the reasons we originally started exploring grid-based space analysis (at the time in 2D) was to be able to perform multiple analyses of the same architectural spaces for the purpose of generative design.

Which leads us to the benefits of using a more discretized approach to 2D or 3D space analysis, via a grid or a voxel model respectively. Firstly, you can adjust the resolution of the discretization which gives you greater control over the trade-off between speed and precision. If you need something to run quickly, you can use larger grid cells or voxels. But you won't have the same precision of results.

Secondly, using a voxel-based approach – and specifically choosing to make sure the result of each analysis is in turn a set of voxels – allows you to combine analyses much more easily. This composability allows you to chain operations together that would have been much harder to combine with a vector-based approach.

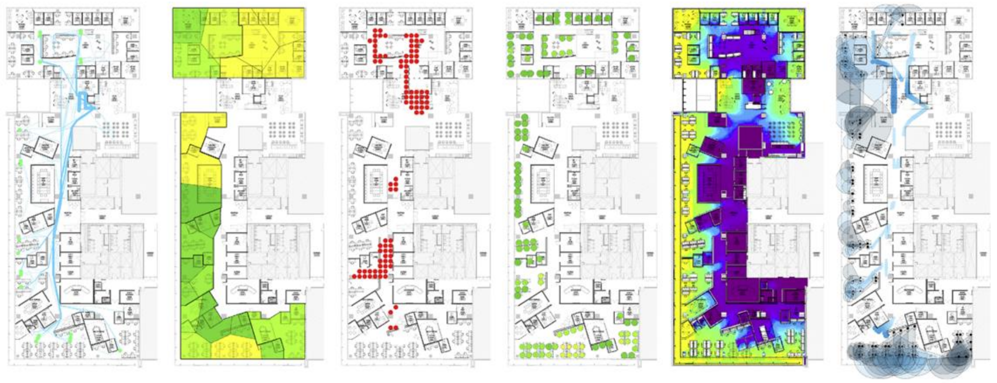


A voxel-based isovist created using in Dynamo using VASA

History of Space Analysis at Autodesk Research

As mentioned previously, Autodesk Research started investigating the use of grid-based space analysis to support generative design workflows. In fact, it was for one workflow in particular: the design of the Autodesk office in Toronto, Canada.

The original Project Discover implemented a number of spatial metrics that assessed the quality of spaces in the future office.



A visualization of the 6 metrics used for Project Discover.

While the original project in 2016 was implemented using a custom workflow, a follow-up project to publish a similar workflow based on Autodesk tools required a Dynamo package that could perform a number of analyses including pathfinding and visibility.

This led to the creation of the 2D Space Analysis package, which used a technique called "central path counting" to perform various analyses on a 2D grid. Here are some resources that talk about some of the approaches adopted in this toolkit:

Path Counting for Grid-Based Navigation
A paper on our space analysis algorithm in the Journal of Artificial Intelligence Research
A Short and Direct Walk with Pascal's Triangle
A Quick and Clear Look at Grid-Based Visibility
Visibility in Space Analysis: an explainer

While analysis of space from a 2D perspective is very valuable, the research team was keen to extend these efforts to 3D scenarios. VASA – which stands for Voxel-based Architectural Space Analysis – is the result of this work.

VASA provides these main capabilities:

- Voxelization of mesh geometry
- 3D pathfinding
- Visibility from a point
- Direct daylight & shadows
- Travel distance from a point
- Surface, solid & fill operations

When working with VASA you typically take source geometry from Revit or Dynamo (perhaps loaded from an STL file) and then voxelize it to have a “voxel model” that supports a variety of operations and analyses.

VASA is a toolkit that allows you to work with voxels: all of the analyses take voxel models as inputs and create voxel models as outputs, which makes them highly composable and flexible.

Even the shortest path between two points in space can be returned as a set of voxels, for instance, which can then be analysed and manipulated using other VASA features.

What’s in a name?

While a convenient acronym, there was an additional reason for choosing the name VASA.

The Vasa was a 17th-century Swedish warship that famously sank on its maiden voyage, without having even left Stockholm’s harbour. It was so encumbered by weapons and ornamentation that it toppled over.

An illustration of the Vasa by Olga Antonenko

It's also a metaphor that has been used in the software industry when talking about overly ambitious projects loaded with features.

Bjarne Stroustrup, the inventor of C++, said this about the Vasa, for instance:

> *Recent research has shown that a relatively modest increase of the Vasa’s length and breadth (claimed technically feasible) would have made it stable, so my reading of the Vasa story is: Work hard on a solid foundation, learn from experience, and don’t scrimp on the testing.*

We took this quote as a source of inspiration in the development of the VASA package – a reminder to ensure that its voxel models served as a solid foundation for a range of operations and analyses.

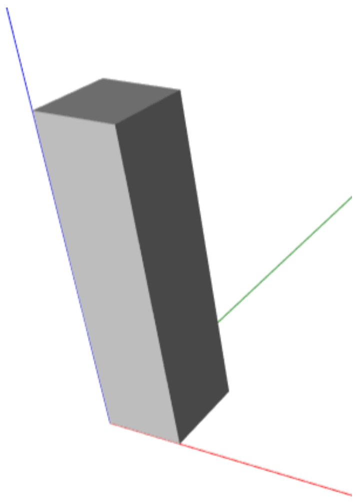
How VASA Works

Let's now take a closer look at VASA's *solid foundation*.

1. **Large models**
    - VASA has been designed to work on buildings and even urban environments, a much larger scale that typical voxel-based systems used in mechanical design, for instance.
2. **Fine resolutions**
    - VASA's resolution has been planned to allow for it to simulate how people move through architectural spaces, navigating narrow doorways, etc.
3. **Low memory**
    - It would have been easy to represent a voxel space as a huge array of Booleans, indicating whether specific voxels were on or off. This would have quickly become unmanageable in terms of the memory required, however, so we have chosen encoding schemes that would allow the system to scale (point 1 above).
4. **Fast algorithms**
    - The intention was always to use VASA in an environment where performance was important, such as when generating and analyzing large quantities of designs. This also means it can work well in interactive environments.
5. **Optimized for buildings**
    - With mechanical models the concept of "up" can be arbitrary. The same is not true at the architectural scale, so we make use of this to introduce efficiencies (points 3 and 4 above).

Voxel sizes

While it's possible to create cubic voxels with VASA, the default size is actually a cuboid:



The default voxel size is 6.25cm x 6.25cm x 25cm

This size of voxel is intended to allow navigation through narrow spaces (a typical "agent" will have a diameter in multiple voxel widths) but allow going up and down stairs and ramps (voxels in a path must be adjacent, so having a height of 25cm allows steps to be considered adjacent).

It's worth noting that voxels must be square in X and Y, despite possibly differing in Z.

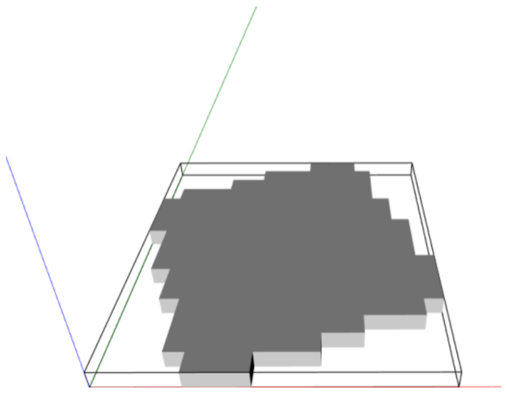
Voxel encoding

As mentioned before, we do not just maintain arrays of Boolean values to represent voxels. They are represented in sets – or rows – of 16 voxels, which can be encoded as a 16-bit integer:

0010111010010110

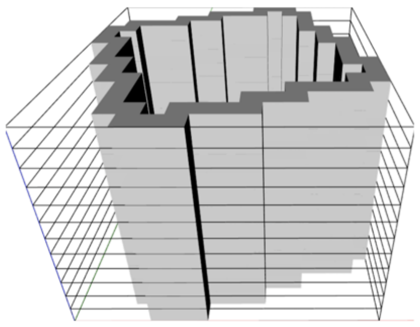
Encoding in this way reduces memory consumption, of course, but also means we can use standard, highly efficient arithmetic operations to implement common algorithms. For instance, a *union* operation between two sets of voxels can be performed using a "bitwise OR" while an *intersection* operation can be implemented as a "bitwise AND".

Rows of voxels are composed into plates of 16 x 16 (represented as 16 integers).



A voxel plate of 16 integers

Plates are then composed into vertical stacks that can be 64 plates high.



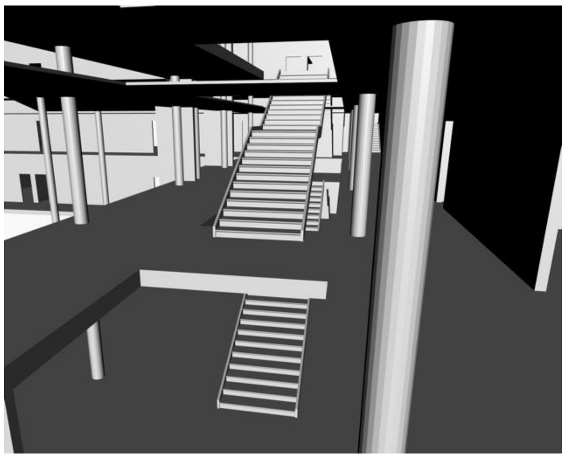
A voxel stack of up to 64 plates

Architectural spaces often have straight walls, a property we can use to our advantage. We can use a technique called run length encoding to minimize memory: if the same plate is repeated in a stack, we simply store the plate and the number of times it repeats. In the above example – which has a solid floor but 11 identical plates representing the walls – we would only need to store two plates and then the number 11 to indicate the second plate is repeated 11 times.

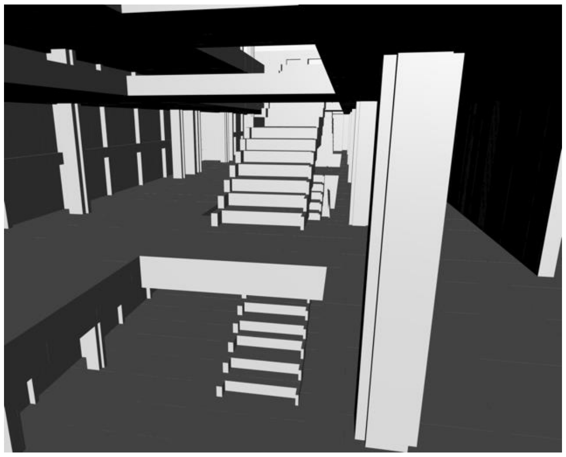
Using voxels for pathfinding

Let's take a look at the steps VASA goes through to find a path between two points in an architectural space.

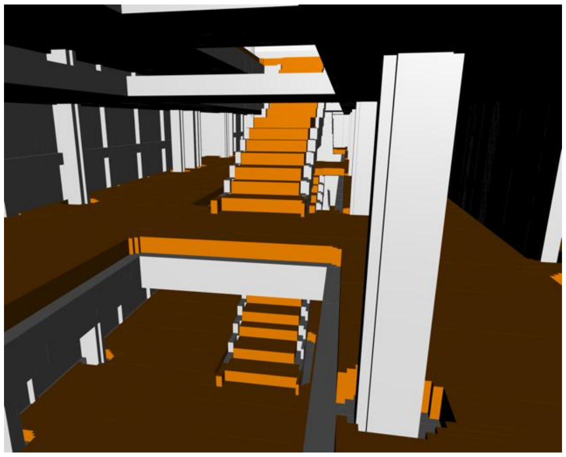
We start with building geometry that might come from Revit or perhaps just from an STL file.


Building geometry

This needs to be voxelized: the geometry is analyzed and if there is some kind of geometry in the voxel location, the flag for this voxel gets turned on.

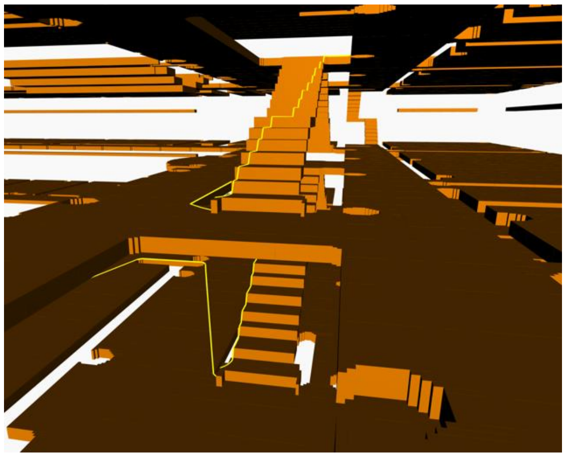

Voxelized geometry

From this basic voxel model, a path model can be created that contains the traversable voxels. These are the voxels that will be used to build a path. It’s a layer of voxels that sits on top of the building voxels.


Traversable voxels

You can see from the above that the steps are voxelized in a way that supports pathfinding: there's clearly adjacency – with no obvious gaps – between stairs.

It's worth noting that there's a gap left around geometry such as the column on the right: the human form doesn't make it likely someone's feet would be right up against a column – even if touching it with another part of their body – so this buffer is left to take this into account. (It's based on the agent diameter setting, which is configurable.)

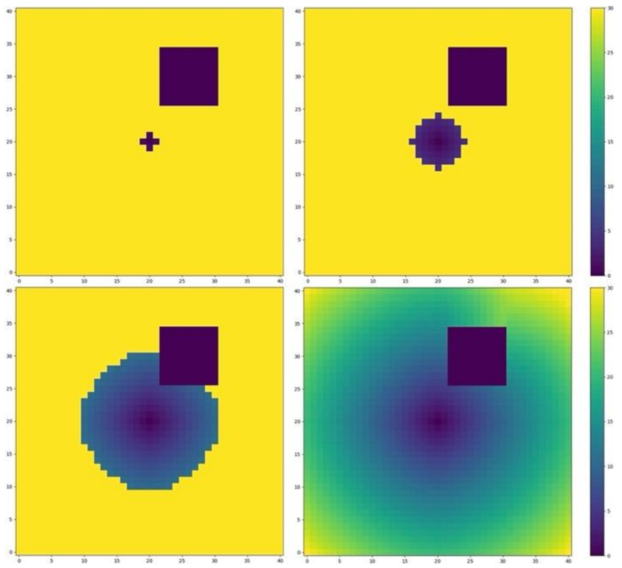

A path found through the model

The traversable voxels are analyzed when searching for a path between two points. In the above image a path has been found that goes up multiple stories of a building. On the lower staircase the path hugs the lefthand side – as the path will loop round on the left of the stairwell – but cuts across the voxels on the upper staircase to reach a point that is somewhere to the right of its top.

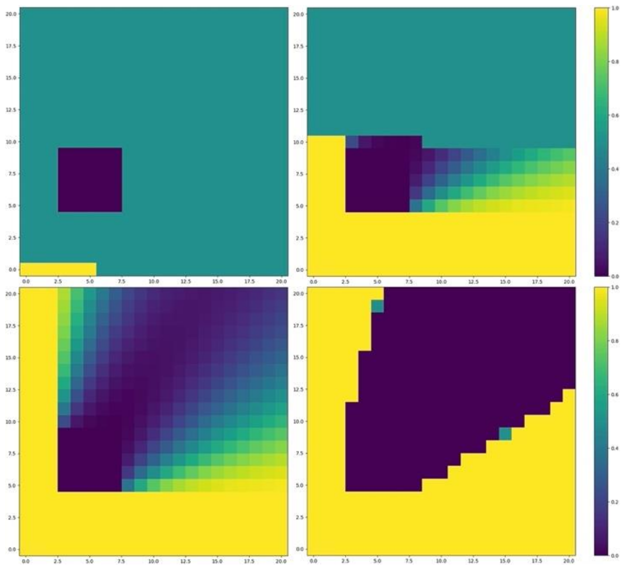
Algorithms for analyzing voxel models

Voxel models provide interesting opportunities to create efficient analysis algorithms.

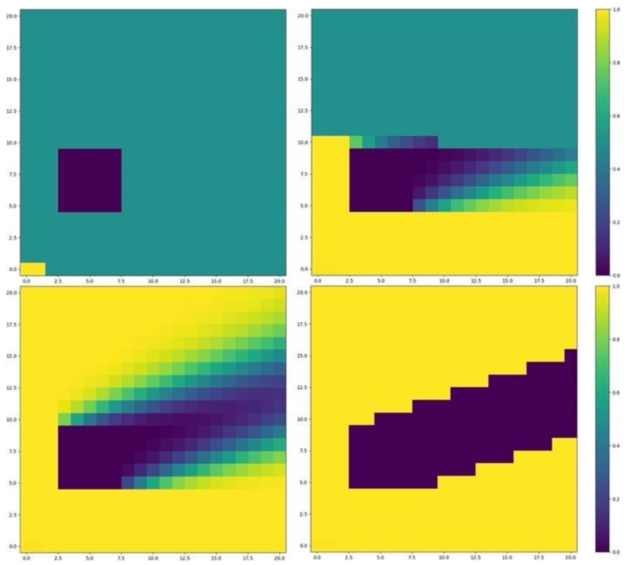
For one we can use a distance field that gradually spreads outwards to find shortest paths between points.


Different stages of pathfinding using a distance field

To calculate visibility from a point – in this case the bottom left of the image – we can calculate the visibility value of each voxel based on its immediate neighbors in the scanning process.

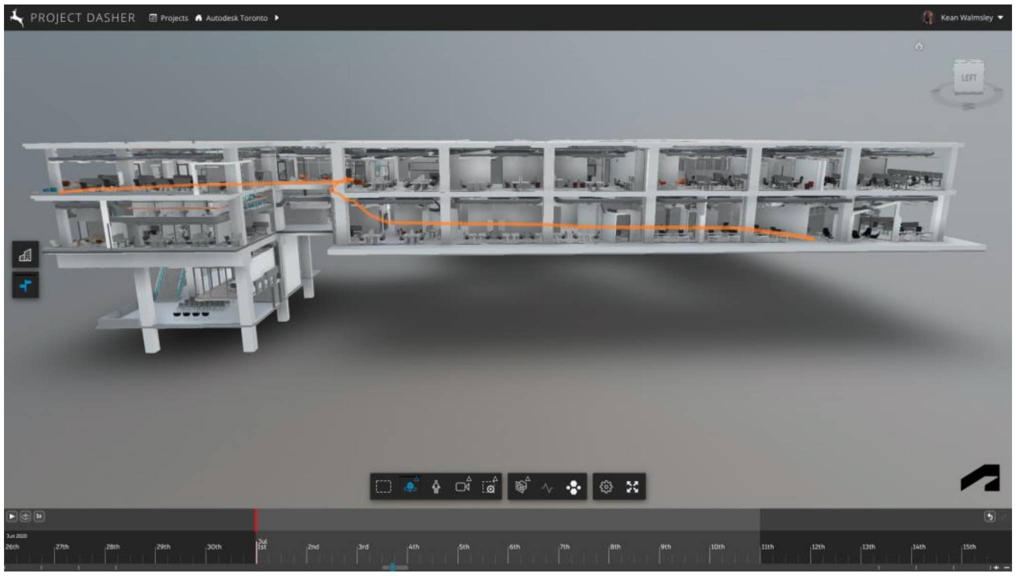

Different stages of calculating visibility from a point.

Similarly for visibility from a direction – which is commonly used to calculate how direct sunlight casts shadows – we can also calculate the value based on neighbours.

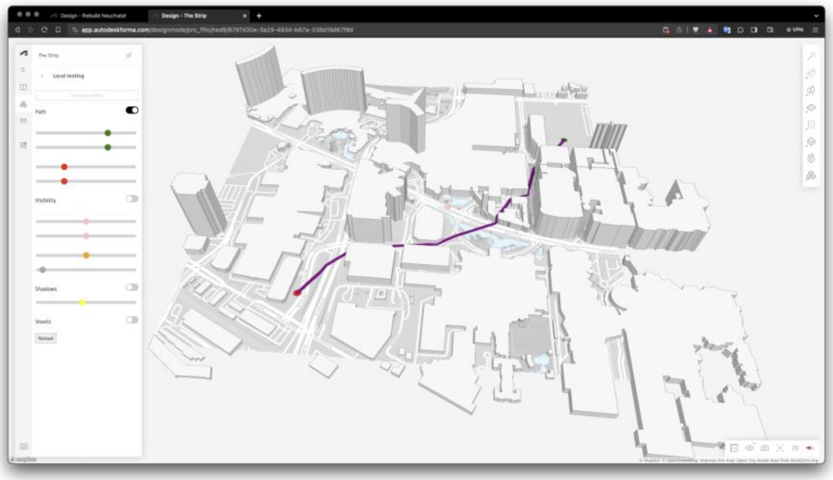

Different stages of calculating visibility from a direction.

Integration with web applications

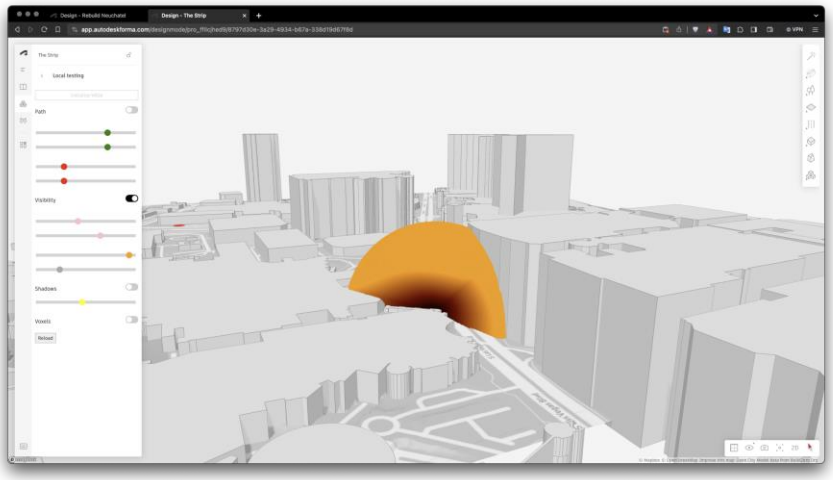
Over the last 2 years, Autodesk Research has been maintaining an internal WebAssembly version of VASA, built using Emscripten. While not available publicly, as a proof of concept this has been integrated into a variety of prototype applications based on the APS Viewer and Tandem SDKs, as well as into a Forma extension.

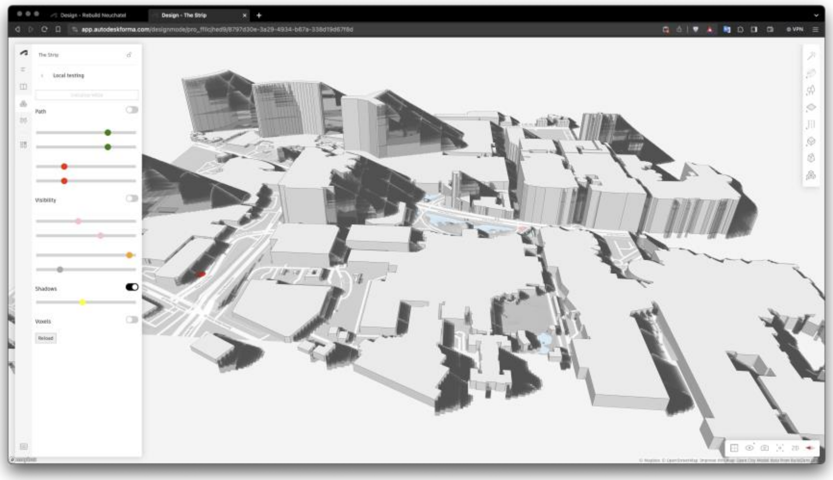


VASA pathfinding working inside a custom web application hosting Autodesk Tandem



VASA pathfinding working inside a Forma extension


VASA visibility from a point working inside a Forma extension


VASA visibility from a direction (for casting shadows) working inside a Forma extension