CindyJS Plugins

Extending the mathematical visualization framework

Martin von Gagern[1⋆,⋆⋆] and Jürgen Richter-Gebert[2⋆⋆]

[1] University of Potsdam, Germany
gagern@uni-potsdam.de,
http://www.math.uni-potsdam.de/professuren/didaktik-der-mathematik
[2] Technical University of Munich, Germany
richter@ma.tum.de,
http://www-m10.ma.tum.de

The final publication is available at link.springer.com
doi:10.1007/978-3-319-42432-3_40

Abstract. *CindyJS* is a framework for creating interactive (mathematical) content for the web. It can be extended using plugins, two of which are presented here.

- *Cindy3D* enables displaying 3D content via WebGL.
- The *KaTeX* plugin typesets formulas within *CindyJS*.

We also discuss the general structure of plugins in CindyJS.

Keywords: Interactive visualization, web technologies, 3D, geometry, WebGL, OpenGL, typesetting, TeX, KaTeX

1 Introduction

The *CindyJS* project is a system for the presentation of visual and interactive mathematical web content (see [8]). It aims to be feature compatible with *Cinderella* [7], a Java-based authoring system. It is possible to extend *Cinderella* using custom plugins. *CindyJS* provides a similar mechanism to allow extension by plugins. Compared to the original Cinderella plugin interface, the plugin api of *CindyJS* offers more possibilities: while a plugin to *Cinderella* is essentially restricted to providing new functions for the built-in scripting language *CindyScript*, plugins in *CindyJS* can perform additional tasks, by accessing selected portions of the internal data of a visualization, like the canvas of the construction or unevaluated expressions passed to a plugin-provided function.

⋆ The author was supported by the project "M C Squared" which has received funding from the EU 7th Framework Programme (FP7/2007-2013) under grant agreement no. 610467.

⋆⋆ The authors were supported by the DFG Collaborative Research Center TRR 109, "Discretization in Geometry and Dynamics".

2 Plugin Interface

At the JavaScript level, a plugin for *CindyJS* is simply a callback method registered with either the *CindyJS* framework as a whole, or one specific widget. That function can interact with *CindyJS* using a set of API functions, the most important of which are probably a function to define new *CindyScript* functions and a function to evaluate a given expression. In order to help maintain backwards compatibility, plugins must declare which version of the API they are using, so that the framework can apply a compatibility layer if its internal representation were to change. Here is a simple example:

```
// Register a plugin called "hello", using plugin API version 1
CindyJS.registerPlugin(1, "hello", function(api) {

  // Define a CindyScript function called "greet"
  // that takes a single argument
  api.defineFunction("greet", 1, function(args, modifs) {

    // Evaluate the argument expression
    // (as opposed to inspecting the unevaluated formula)
    var arg0 = api.evaluate(args[0]);

    // Return string as a CindyScript value object,
    // we might want to offer some API for this one day
    return {
      ctype: "string",
      value: "Hello, " + api.instance.niceprint(arg0)
    };

  });
});
```

3 Viewing spatial objects using *Cindy3D* and WebGL

The goal of the *Cindy3D* plugin is visualizing spatial mathematical objects using WebGL.

Cinderella (the Java ancestor of *CindyJS*) has a plugin of that same name [6], based on JOGL to provide OpenGL bindings. The *Cindy3D* plugin for *CindyJS* started out as a port of that Java plugin, but by now most of the code has been rewritten, so the main connection is a common API that is exposed to *CindyScript*. In the long run, it is expected to port large ports of the plugin back to the Java version, for consistent results, easier maintainance and common sets of features. In the subsequent text, the term *Cindy3D* will refer to the *CindyJS* plugin only.

Cindy3D features four kinds of geometric primitives: spheres, rods, polygons and meshes. A sphere is often used to denote a point in a 3D setup, just as

```
// Set up 3D environment
begin3d();
background3d([0,0,0]);
size3d(2.4); // Default size of points and segments

// Declare some constants which may be used to tune appearance
n = 300; r1 = 1; r2 = 0.3; k = 5; l = 3;

// Compute point for given parameter value w
f(w):=(sin(l*w), cos(l*w),           0)*(r1 + r2*cos(k*w))
    + (       0,        0, r2*sin(k*w));

// Connect consecutive points with a colored rod
repeat(n, i,
    w1 = i/n*360°;
    w2 = (i+1)/n*360°;
    color3d(hue(i/n));
    draw3d(f(w1), f(w2));
);

end3d();
```

Fig. 1. Code and result for a simple 3D object

disks denote points in the planar view of *CindyJS*. A rod is a cylinder with two spherical endpoints, and therefore the 3D counterpart to a line segment. A polygon is a planar surface (although it's the user's task to ensure that the vertices actually lie within a single plane). Non-planar surfaces are modeled as meshes, which are triangle meshes internally but quadrilateral meshes in the *CindyScript* API.

Contrary to many other 3D rendering environments, spheres and rods are not subdivided into triangle meshes. Instead, an object which covers the sphere or rod is drawn with a custom fragment shader to render a high-fidelity representation of the actual geometric object using a raycasting approach. In the case of a sphere, the covering object is a square facing the camera and just in front of the sphere. For the rod, the covering object is a box containing the actual rod. One consequence of this rendering approach is that the position of the geometric primitive being rendered does not correspond to the position of the final point once it's rendered: the depth may differ, so the shader code has to update the fragment depth. This requires the use of a WebGL extension called `EXT_frag_depth` which isn't available on all devices yet, although the percentage of supported devices is ever increasing.

Cindy3D employs some simple raycasting to provide cheap yet realistic lighting of the scene. The details of the appearance can be controlled by placing lights, by controlling object material properties like color or shininess, and of course by placing the camera and configuring its lens.

Cindy3D supports translucent objects, which is important for many mathematical visualizations. This is a challenging task, since for accurate results the scene has to be rendered strictly from back to front. As of this writing, *Cindy3D* doesn't sort its primitives yet. But it already represents all its triangles as distinct objects, which greatly simplifies the task of back-to-front rendering, since the triangles of different meshes can then be sorted as a whole, giving correct order in regions where both meshes twist around one another. Even more demanding would be the task of accurate back-to-front rendering in places where the constituent triangles intersect. That would require computing the corresponding intersections. No such endeavur is planned for the near future. In commonly used examples it is often surprising how close one has to look to spot errors due to the incorrect rendering order of the current naive implementation.

Cindy3D is not designed to allow manipulation of displayed objects. The objects are constructed from a sequence of drawing commands in *CindyScript*, and then viewed in *Cindy3D* as they are. What *can* be controlled interactively is the configuration of the camera. It can be rotated around the object, which for every point and purpose is the same as rotating the object around the look-at point of the camera since light sources can be fixed to the camera or the object at the user's discretion. It can also be rotated around itself, or translated in three spatial directions. The camera can move closer to the object, or farther out, which is colloquially called a zoom. It can also perform an actual photographic zoom, i.e. change the field of view. All of these operations are accessible by mouse movements, in combination with modifier keys.

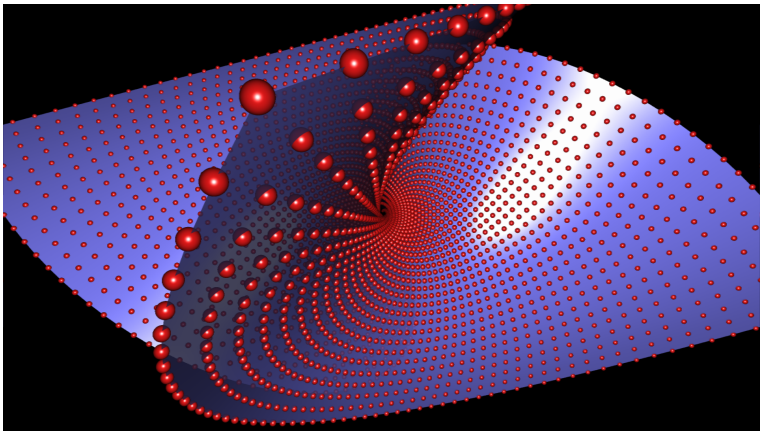

Fig. 2. Translucent Enneper surface with spheres indicating grid points

The source code of *Cindy3D* is clearly structured into code which provides specific bindings to the *CindyJS* API, and code which does the internal data representation and manipulation. It would be fairly easy to replace the former by bindings for some other software package, in order to turn *Cindy3D* into a 3D model viewer for a different web-based (or at least browser-based) application.

4 Typesetting formulas using *KaTeX* bindings

Some mathematical content can be best described using a combination of geometric elements and formulas. Sometimes it is enough to include the formulas in the text surrounding a given widget. But if the text has to be positioned with respect to a given element of the widget, or perhaps contains numbers which change during interaction, then it is important to typeset these formulas within the widget.

The lingua franca for entering formulas is TEX for most mathematical communities. *Cinderella* comes with its own home-grown parser and renderer for TEX-like formulas. For *CindyJS* it was decided not to port that parser, but instead build on one of the existing efforts to bring math typesetting to the web. While MathML has been intended as default representation for formulas, browser support for it is severely lacking, with no change anticipated in the near future.

The most common sollution is *MathJax* [3], a JavaScript library to render formulas. But *MathJax* has several problems which make it unsuitable for the application at hand. It operates on the HTML DOM tree, so one would have to somehow position the text above the widget, instead of drawing it to the widget canvas the same way geometric objects are being drawn. It operates asynchroneously, so there is a delay between the time when drawing a formula is requested and the time when the typeset version of said formula is actually

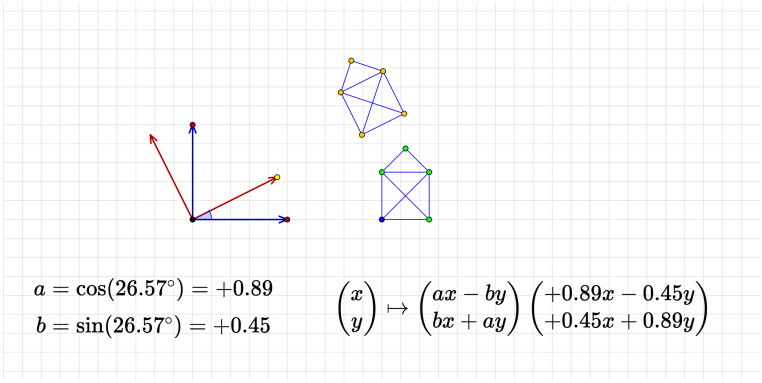

Fig. 3. Educational widget using typeset formula

ready for display. This fits in poorly with the synchroneous drawing paradigma employed by *CindyScript*.

Looking for an alternative, we found the *KaTeX* project [1]. It provides synchroneous operations, and usually renders significantly faster than *MathJax*. The main drawback is its lack of features: many things supported by *MathJax* are (or at least were) not available in *KaTeX*. We identified those features whose absence would cause the most trouble for existing or envisioned content, and had those features implemented for *KaTeX*. Foremost on that list is support for matrices, which was developed for *CindyJS* but has been merged into the official *KaTeX* code base as well.

Just like *MathJax*, *KaTeX* is designed to modify the HTML DOM tree. But it has an internal intermediate representation of how to arrange and nest its boxes, from which the actual HTML elements are being generated. Using this representation it was possible to modify the code in such a way that instead of creating HTML elements it creates canvas drawing commands. Since the internal representation only provides vertical placement information, horizontal positioning has to measure text dimensions. For this reason, the render-to-canvas process has two phases. In the first phase, the individual blocks are measured and positioned relative to one another. The result is an object which contains enough data to render all required glyphs, but which also has overall measurements of the whole formula. These can be used to position the box, e.g. for horizontal alignment, before the glyphs get actually drawn to canvas. The draw-to-canvas feature hasn't been merged into the official *KaTeX* (yet?), mainly since it caters for some very specific use cases only. Nevertheless, this feature can be of use to other projects facing the same problem of how to place high-quality math typesetting in a web application based on the HTML canvas element.

The *KaTeX* plugin for *CindyJS* is a fairly thin layer binding *CindyJS* to a version of the *KaTeX* library which includes our customizations like render-to-canvas. It is different from other plugins in that it doesn't provide any new *CindyScript* commands. Instead it modifies the behavior of existing commands

using a hook in the *CindyJS* rendering pipeline. Much of the complexity of the plugin, however, is spent on ensuring the automated loading of required resources. Fonts in particular are difficult to handle, since on some browsers loading of these only starts once they are actually being used, so they won't be available upon that first use. In this situation, the *KaTeX* plugin will not render the text but instead wait for the required fonts and trigger a repaint once they become ready.

5 Other plugins

There have been other successfull applications of the plugin infrastructure as well. The following list demonstrates the high flexibility of the plugin infrastructure.

CindyGL is a tool which allows running a subset of the *CindyScript* language on the GPU. It is described in a separate article, [5].

QuickHull3D is an algorithm [2] and a Java library [4] used by *Cinderella* to implement its `convexhull3d` operation. That Java code was compiled to JavaScript using GWT, and made available as a plugin, in order to provide compatibility. This is a temporary solution since it would be preferable to have a native JavaScript implementation inside the core of *CindyJS*, which is something currently being worked at. However, this setup demonstrates that plugins can be used to build connections even to some Java libraries.

MC Squared (or Mathematical Creativity Squared) is a project which allows authors to combine widgets from various sources into so-called C-Books. When viewing them in a HTML5 environment, *CindyJS* is used to display *Cinderella* widgets, and a plugin is used to realize the integration into that specific environment, in particular the communication with other widgets.

Metadata extraction from images generated by the ornament drawing app *iOrnament* was demonstrated in a proof-of-concept plugin implementation. This made it possible to post-process the ornaments in a way compatible with the symmetry group used for their creation.

Tests in the *CindyJS* test suite sometimes use plugins to allow a *CindyJS* instance to report results back to the testing framework.

Development of the complex tracing implementation within *CindyJS* itself was helped by visualizations of the tracing process. One instance of *CindyJS* was showing some construction to be interacted with, while a second instance turned debug logs of the operations behind the scenes into helpful visualizations, providing a real-time view of the corresponding internal computations. Those logs were generated by the first instance and made available to the second via a custom plugin.

6 Conclusion

Plugins are a useful way to extend a software framework in order to adapt it to new requirements. Often the people writing the plugins are distinct from

those writing the core framework. So far, *CindyJS* has not attracted any third party plugins that we know of. But the fact that plugins are being used by the developers themselves ensures that the plugin infrastructure is powerful and flexible enough to accomodate various requirements. They are particularly useful for connecting optional components that provide their own interfaces, translating between different conventions and representations. Plugins are the perfect tool for extending functionality for custom applications without bloating the core implementation for everyone.

References

1. Ben Alpert and Emily Eisenberg. Katex. `https://khan.github.io/KaTeX/`, 2013.
2. C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.
3. Cervone Davide, Volker Sorge, Christian Perfect, and Peter Krautzberger. Mathjax. `https://www.mathjax.org/`, 2009.
4. John Lloyd. Quickhull3d – a robust 3d convex hull algorithm in java. `http://www.cs.ubc.ca/~lloyd/java/quickhull3d.html`, 2004.
5. Aaron Montag and Jürgen Richter-Gebert. CindyGL: Authoring GPU-based interactive mathematical content. unpublished. Submitted to ICMS 2016 Berlin.
6. Matthias Reitinger and Jan Sommer. Cindy3d. `http://gagern.github.io/Cindy3D/`, 2012.
7. J. Richter-Gebert and U. Kortenkamp. *The Cinderella.2 Manual: Working with The Interactive Geometry Software*. Springer, 2012.
8. Martin von Gagern, Ulrich Kortenkamp, Jürgen Richter-Gebert, and Michael Strobel. CindyJS – Mathematical visualization on modern devices. unpublished. Submitted to ICMS 2016 Berlin.