CindyGL: Authoring GPU-based interactive mathematical content

Aaron Montag and Jürgen Richter-Gebert*

Technical University of Munich, Germany
{montag,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_44

Abstract. *CindyJS* is a framework for creating interactive (mathematical) content for the web. The plugin *CindyGL* extends this framework and leverages WebGL for parallelized computations.
CindyGL provides access to the GPU fragment shader for *CindyJS*. Among other tasks, the plugin *CindyGL* is used for real-time colorplots. We introduce the main principles, concepts and application of *CindyGL* and describe the encountered technical challenges. Special focus is put on a novel visualization scheme that uses feedback loops, which were among the motivating forces of developing CindyGL. They can be used for a wide range of applications. Some of them are numerical simulations, cellular automatons and fractal generation, which are described here.

Keywords: Interactive visualization, web technologies, WebGL, transpiler, CindyScript, GLSL, OpenGL, shader based colorplots, feedback loops on GPU, fractals, limit sets, IFS, Kleinian groups

1 Introduction

The *CindyJS* project is a system for authoring dynamic mathematical web content (see [7]). It allows web based prototyping of mathematical experiments and visualizations which can be used for research and demonstration. *CindyScript* is a scripting language for *CindyJS*, that can be directly used in the HTML code. For the design principles of *CindyScript* we refer to [4]. Its language specifications are presented in the Cinderella 2 handbook [5].

In this article the plugin *CindyGL* for *CindyJS* is introduced. *CindyGL* is a plugin for *CindyJS* which provides the high-level mathematically oriented user with access to the shader language of the GPU.

In most other scenarios, knowledge of JavaScript and a shader language is required and many lines of "boilerplate-code" have to be written in order to build even small shader examples in WebGL. On the other hand, they often

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

could be described with only few words. One aim of the WebGL integration into *CindyScript* through the *CindyGL* plugin is overcoming the technical obstacles that are typically inevitable in usage of OpenGL technologies on the web. While writing *CindyScript* code, the user should not even become aware of using WebGL.

The second aim of *CindyGL* is providing a simple fast-prototyping tool for *feedback loops on the GPU*, that can be used for various novel algorithms. No other web project that overcomes both of this difficulties is known to us.

The technical core of this plugin is a transcompiler, which can translate *CindyScript* to OpenGL Shading Language (GLSL). Aside general-purpose computations on the GPU, the transcompiler is so far used for rendering 2D-colorplots on the GPU. If required, the 2D-colorplots can be animated in real-time as well.

This function is accessible via the `colorplot` command of *CindyScript*. A set of running examples (with their source code) can be viewed on `http://cindyjs.org/CindyGL/`.

2 The colorplot command

The *CindyScript* primitive operations for accessing *CindyGL* were designed such that the boiler plate for creating WebGL-applications is minimized. For instance, an animated plot of the interference of two circular waves can be rendered by evaluating the following *CindyScript* code at every animation step:

```
t = seconds() - t0;
colorplot(
  (sin(|A,#|-t) + sin(|B,#|-t)+2) * (1/2, 1/3, 1/4)
);
```

A static image of the animated result is depicted in Figure 1 (a). The expression inside the `colorplot` command maps pixel coordinates (at position `#`) to colors (encoded as a 3-component rgb vector). Here `|A,#|` and `|B,#|` are the distances between the current pixel coordinate and the two points on the *CindyJS* canvas, that can be interactively repositioned by drag and drop.

A phase portrait for the complex function $f : \mathbb{C} \to \mathbb{C}$, $z \mapsto z^7 - 1$ can be rendered as follows (the concept of complex phase portraits is explained in [9]):

```
f(z) := z^7 - 1;
colorplot(
  hue(im(log(f(complex(#)))) / (2*pi));
);
```

This program outputs a GPU rendered image as in Figure 1 (b). The argument of `f(complex(#))` determines the color for the pixel with the coordinate `#`. Note that the computation of complex numbers was inherently carried to the GPU, which has no native support for complex calculus.

Furthermore, sophisticated colorplots are possible as well. As an example, a raycaster for algebraic surfaces can be written as a colorplot in *CindyScript* as depicted in Figure 1 (c). Following the approach of [6], we compute the intersection of each view ray with the algebraic surface as the root of a polynomial that is determined by an interpolation process. In *CindyScript* the interpolation – a linear function that maps lists of evaluated function values to a vector of polynomial coefficients – can be easily described as a matrix multiplication. This matrix computation is coded high-level in *CindyScript* and transpiled to the GPU.

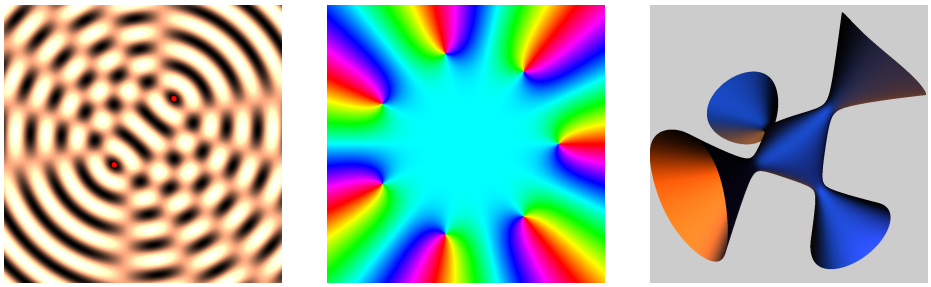

Fig. 1. Screen shots of animations generated by the `colorplot` command: (a) Interference of waves, (b) a complex phase portrait and (c) a raycaster for algebraic surfaces

3 Feedback Loops via *CindyGL*

The `colorplot` command was designed in a way that it is possible to write on textures by a function passed as an argument. The textures then in turn can be read in consecutive calls of `colorplot` with the `imagergb`-command. The possibility to read and write texture data, immediately enables the creation of *feedback loops on the GPU*.

By the term *feedback loop* we mean a system of a set of images that are iteratively re-generated by using themselves. A "physical" example of a single-image feedback loop is the "infinite tunnel" that becomes visible if one points a camera at a screen which directly displays a live video recorded by the camera.

An example for a *feedback loop* in *CindyGL* can be seen here:

```
colorplot("julia", // plots to texture "julia"
 z = complex(#);   // if |z| < 2, take the color from texture "julia"
 if(|z|<2,         // at position z^2 + c and make it slightly brighter
  imagergb("julia", z^2+c) + (0.01, 0.02, 0.03),
  (0, 0, 0)        // if |z| ≥ 2, (z, f(z), f^2(z), ...) is not bounded;
 )                 // display black.
);
```

If the code is executed several times, a picture of a Julia-fractal for the function $f : z \mapsto z^2 + c$ progressively emerges on the texture `julia`. After roughly 50 iterations a picture as in Figure 2 (a) becomes visible. The Julia-fractal for a specific function depicts the points which remain bounded if the function is iteratively applied to them. The fact that only one iteration per pixel for each rendering step is computed makes the rendering process very fast and enables a real-time escape-time based fractal visualization. Here, a direct and smooth interaction with users changing the parameter c on the fly is possible, even on mobile devices.

Figure 2 (b) shows a picture of Conway's Game of Life, where cells of a 2-dimensional grid can either be alive or dead. In each computation step, a cell can die or be reborn according to the number of its living neighbors. Using black (`0`) and white (`1`) pixels for dead and living cells respectively, this cellular automaton can be simulated with *CindyGL* as follows:

```
// a function that reads the 80x80 texture "gol",
// assuming a torus like world.
get(x, y) :=  imagergb("gol", (mod(x, 80), mod(y, 80))).r;
newstate(x, y) := ( number = // number of living neighbors
 get(x-1, y+1) + get(x, y+1) + get(x+1, y+1) +
 get(x-1, y) +                 get(x+1, y) +
 get(x-1, y-1) + get(x, y-1) + get(x+1, y-1);
 if(get(x,y)==1, // if the cell lives and it has less than 2
                 // or more than 3 neighbors, it will die.
  if((number < 2) % (number > 3), 0, 1),
                 // if a cell was dead, then 3 neighbors
                 // are required to be born.
  if(number==3, 1, 0)
 )
);

colorplot("gol", newstate(#.x, #.y)); //plots to texture "gol"
```

Here a texture `"gol"`, which encodes the previous state, will be reused as a basis for the computation of all the new states, which will be written to the texture `"gol"` again.

Figure 2 (c) shows a simulation of a reaction-diffusion system using feedback loops. It serves as an example how numerical simulations of 2-dimensional partial differential equations can be computed in real time on the GPU. In this example, and also many other numerical simulations, a very fine time discretization is demanded. Since a single iteration step utilizing a feedback loop construction can be computed very fast on the GPU, many iterations of the feedback loop can be done before displaying a single frame. On today's average hardware, decent frame rates are still possible.

Feedback loops also give a natural framework to render limit sets on the GPU. Visualizations of the limit sets of two dimensional *iterated function systems*

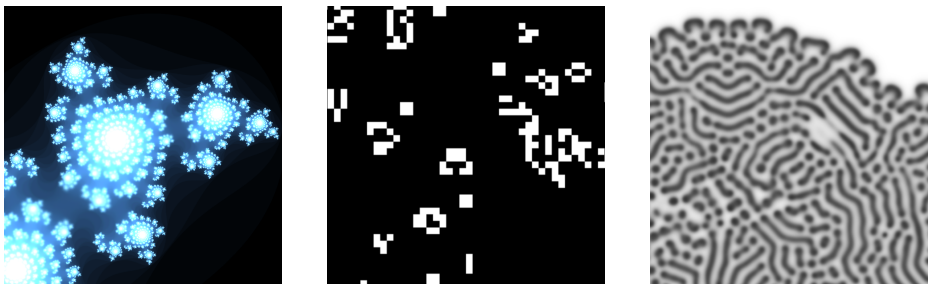

Fig. 2. Visualizations generated by feedback loops: (a) a progressively built up Julia fractal, (b) Conway's Game of Life and (c) a reaction-diffusion model

(IFSs), which are described in [1], can be generated by iteratively applying a slightly modified Hutchinson operator to a texture: A texture is iteratively rebuilt as a composition of deformed copies of itself. This can be considered as a feedback loop of a single texture and results of *CindyGL* implementations are shown in Figure 3 (a) and (b).

By extending the feedback loop system containing a single texture to a system containing multiple textures that are linked in a sophisticated manner, it is also possible to visualize limit sets of certain Kleinian groups in real time. An example image of such an CindyGL generated limit set of a Kleinian group is depicted in Figure 3 (c). The required techniques are derivated and described in detail in [2]. Summarizing the generation of Figure 3 (c), two Möbius transformations were chosen by "grandma's recipe" from [3] to generate a free group. Then a deterministic finite automaton was built that accepts the regular language of the geodesic words of the language of the free group, i.e. the shortest words consisting of the two generators and their inverses describing the group elements. By transferring the states of this automaton to textures and the transition between them to corresponding Möbius transformations that are used to generate each of these textures, a complex interlinked system of textures is generated. Now by iterating simultaneously the generation of these textures, one can prove that in the limit an image of the limit set of the Kleinian group is attained.

CindyGL is a tool that can be used to built such interlinked systems of textures with relatively little effort.

4 Technical Aspects

CindyJS is licensed under the Apache 2 license and can be obtained from `https://github.com/cindyjs`. The plugin *CindyGL* is integrated into the *CindyJS* project.

One development aim for *CindyGL* was obtaining a performance that is comparable with the one of native WebGL applications. During real time animations, the `colorplot` command is called many times within a second. Typically, the

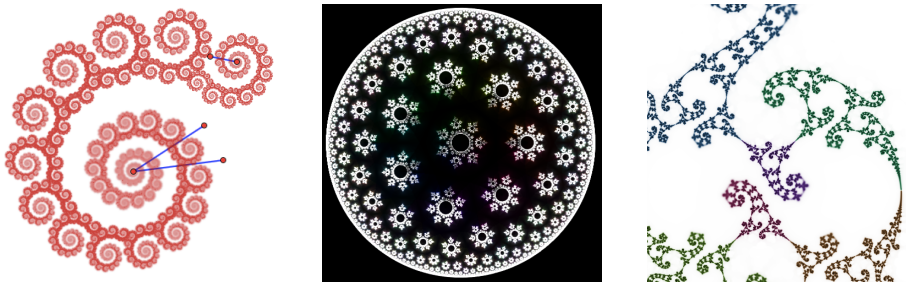

Fig. 3. Images of different limit sets generated by feedback loops: (a) an IFS generated by two affine transformations, (b) an IFS generated by circle inversions and (c) a Kleinian group

syntactic expression within the argument remains the same – only the values of variables might change.

Performance preservation was mainly achieved by doing all the computations that are demanded by an additional layer between a native WebGL application and a *CindyJS* application only at the first time the `colorplot`-command is called. During successive calls of `colorplot` (with the same arguments), a native shader program is executed.

OpenGL Shading Language (GLSL) is the language which is used to run specific programs on the GPU in WebGL. It is strongly typed. In contrast, *CindyScript* has dynamic typing.

We have developed a transcompiler that is able to translate *CindyScript* primitives into GLSL. During this process types of terms and variables in *CindyJS* (e.g. real numbers, complex numbers, matrices, ...) are – if possible – automatically detected and modeled to corresponding data structures on the GPU (e.g. `float`, `vec2`, `mat4`, ...).

A partial order on the types has been introduced in order to capture subtype relations between types. A type is defined to be a subtype of another type (for instance, real numbers are a subtype of complex numbers), if there is a inclusion function from values of the subtype to the other type such that every function having multiple signatures for different types commutes with all the inclusion functions. Hence, the "weakest possible" type can always be chosen for the calculations in order to save resources and obtain good performance.

When the function `colorplot` is called for the first time, the syntax tree of the color expression and functions that are called within this expression are traversed recursively in order to find out the terms that depend on the varying pixel variable `#`. Those terms are suitable for a massive parallelization on the GPU and are translated to GLSL via the introduced transcompiler. A fragment shader is built in WebGL, that computes the corresponding expressions for each pixel, while the other terms that are independent from `#` are calculated just once on the CPU and passed to the GPU as uniforms. Since the segmentation in parallelized code and CPU code is created automatically, this on the on hand

eases the work of the programmer. On the other hand, it very often creates the most general and performant split of the code.

A *ping-pong approach* is used for the feedback loops. If `colorplot` tries to read and write on a texture at the same time, the texture will be stored twice: One texture for reading and another target texture for writing. After the function call, the two textures will be swapped. In the next call of `colorplot` then recently written texture can be used as input texture.

5 Conclusion and outlook

Overall, the *CindyGL* project aims to provide an easy-to-use technical backbone for a wide range of different mathematical visualizations.

CindyGL is not finished yet. Some primitive operations and data structures from *CindyScript* are still missing. Also, an integration of the ideas of [8] is planned. In particular, enabling live access to a camera is possible. Using a live image of a camera-picture as input texture for a `colorplot` opens the door for new educational concepts. The image can be easily deformed using *CindyScript*. An analog design setup where a camera points to the currently displayed image can be used to explain the concept of feedback loops. For a valuable educational experience, the real world, that might consist of persons, patterns or a system of mirrors for example, can be included in the setting. Results of a prototypical setting are shown in Figure 4.

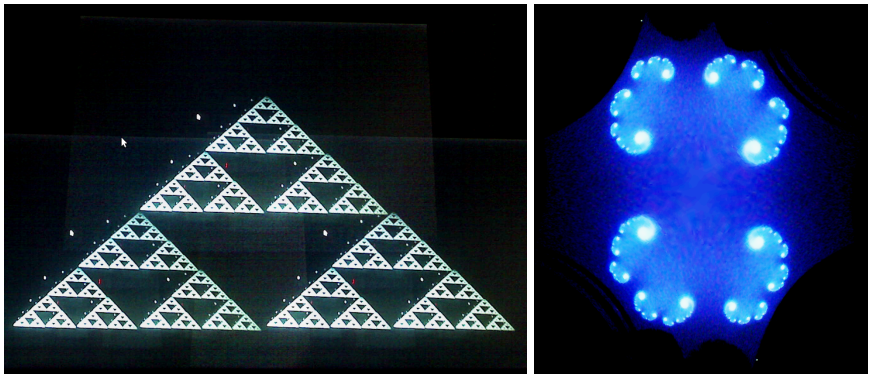

Fig. 4. Two fractals generated by analog feedback loops. (Using an integrated webcam and a mirror)

References

1. Michael F Barnsley. *Fractals everywhere*. Academic press, 2014.

2. Aaron Montag. Interactive image sequences converging to fractals. Bachelors Thesis. Available at `http://aaron.montag.info/ba/main.pdf`.
3. David Mumford, Caroline Series, and David Wright. *Indra's pearls: the vision of Felix Klein*. Cambridge University Press, 2002.
4. J. Richter-Gebert and U. Kortenkamp. The power of scripting: DGS meets programming. *Acta didactica Napocensia*, 3(2):67–78, 2010.
5. J. Richter-Gebert and U. Kortenkamp. *The Cinderella.2 Manual: Working with The Interactive Geometry Software*. Springer, 2012.
6. Christian Stussak. *Echtzeit-Raytracing algebraischer Flächen auf der GPU*. PhD thesis, Diploma thesis, Martin Luther University Halle-Wittenberg, 2007.
7. Martin von Gagern, Ulrich Kortenkamp, Jürgen Richter-Gebert, and Michael Strobel. CindyJS – Mathematical visualization on modern devices. unpublished. Submitted to ICMS 2016 Berlin.
8. Martin von Gagern and Christian Mercat. *Mathematical Software – ICMS 2010: Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, chapter A Library of OpenGL-Based Mathematical Image Filters, pages 174–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
9. Elias Wegert. *Visual complex functions: an introduction with phase portraits*. Springer Science & Business Media, 2012.