Shape Display Shader Language (SDSL): A New Programming Model for Shape Changing Displays

Christian Weichel
Lancaster University, UK
c.weichel@lancaster.ac.uk

John Hardy
Lancaster University, UK
john@highwire-dtc.com

Jason Alexander
Lancaster University, UK
j.alexander@lancaster.ac.uk

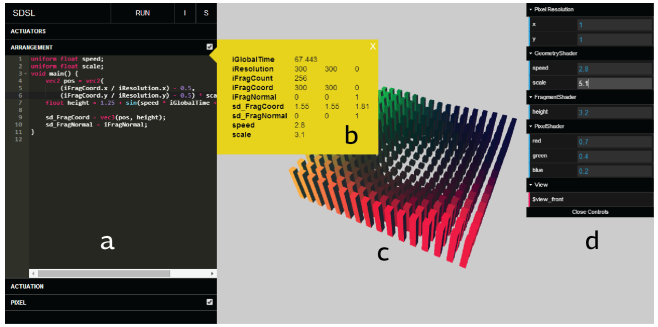

Figure 1: The SDSL IDE. (a) The arrangement, actuation and pixel shader editors. (b) The realtime program inspector. (c) Program running in the simulator. (d) Uniform control panel.

CHI'15 Extended Abstracts, Apr 18-23, 2015, Seoul, Republic of Korea
ACM 978-1-4503-3146-3/15/04.
http://dx.doi.org/10.1145/2702613.2732727

Abstract
Shape-changing displays' dynamic physical affordances have inspired a range of novel hardware designs to support new types of interaction. Despite rapid technological progress, the community lacks a common programming model for developing applications for these visually and physically-dynamic display surfaces. This results in complex, hardware-specific, custom-code that requires significant development effort and prevents researchers from easily building on and sharing their applications across hardware platforms. As a first attempt to address these issues we introduce SDSL, a Shape-Display Shader Language for easily programming shape-changing displays in a hardware-independent manner. We introduce the (graphics-derived) pipeline model of SDSL, an open-source implementation that includes a compiler, runtime, IDE, debugger, and simulator, and show demonstrator applications running on two shape-changing hardware setups.

Author Keywords
Shape-Changing Displays; Shader Programming

ACM Classification Keywords
H.5.2. [Information Interfaces and Presentation (e.g. HCI)]: Miscellaneous

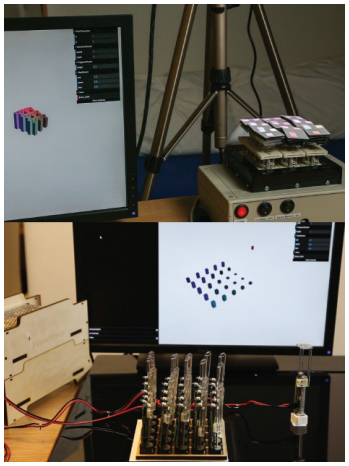

Figure 2: (left) wave animation running in the simulator and on Tilt Display, (right) the same wave animation running on different hardware.

Introduction

Recent interest in shape-changing displays—visual output surfaces that can physically deform—has intensified, with many point-designs showcasing a variety of features, technologies, and applications [1, 5, 10]. To control the display and actuation, each deployment requires a custom, hardware-dependent software architecture. This software requires significant programming effort and is 'unsharable' across devices. The community lacks a common programming model, and vocabulary, for easily creating hardware-independent, portable applications for shape-changing displays.

As a first approach to overcome these issues, we introduce the Shape Display Shader Language (SDSL): a programmable shader pipeline inspired by modern graphics pipelines [3], for developing content for shape-changing displays (motion design). SDSL is a platform-agnostic, hardware-independent pipeline that unifies content development by providing a common programming vocabulary and environment. The SDSL pipeline supports scalable, flexible, and more controllable applications compared to custom-built, hardware-specific solutions. Using mature shader concepts, SDSL allows simple code to achieve complex display, actuation, and interaction scenarios. We provide an initial SDSL implementation for z-actuated displays. SDSL supports our vision of a community who can easily design and share shape-changing applications across different hardware platforms.

This paper contributes: 1) The concept of applying graphics shaders to shape-changing displays; 2) A programming language and IDE, including a simulator and debugger, that support the SDSL pipeline; 3) A demonstration of the hardware agnostic runtime and SDSL flexibility using two different shape-changing displays; 4) An open-source implementation of the SDSL compiler, runtime, and IDE.

Related Work

Shape-changing displays encompass a diverse range of physically-dynamic visual output surfaces. These devices can be described by their topological form, transformation, and interaction [8]; their materiality or surface properties [2]; or by their geometric shape [10]. SDSL represents shape-change using geometric modeling of surface features.

SDSL provides programming support for single-coordinate space shape-change. This allows it support many of the prototypes present in the literature: z-actuation [5, 7], tilting [1], and bending [10]. Due to the adaptable nature of the pipeline, material properties such as stiffness control [4] could also be included.

In modern computer graphics, programmable shader pipelines [3] enable a high degree of control over the rendering process. Shaders are written in hardware-independent languages (e.g., GLSL [9] or HLSL [6]) that allow developers to focus on content presentation. Existing graphics shader pipelines can be re-purposed for shape-changing displays [5] to leverage the parallelism of graphics hardware. However, this requires a lot of custom code and is hardware specific. Previous work has also applied shader pipelines to non-graphic domains. OpenFab [11] is a programmable pipeline for multi-material 3D printing. Much like SDSL, OpenFab defines it's own shader types and programming language.

SDSL

Using a shader-based programming model for shape-changing displays eases development, and makes

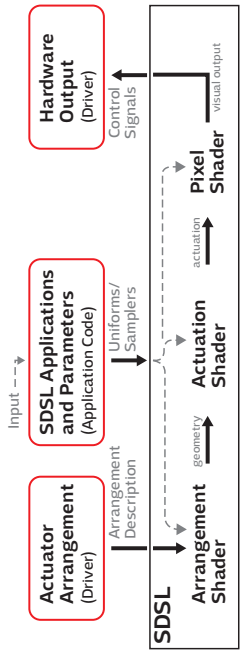

Figure 3: The SDSL pipeline, including arrangement shader, actuation shader, and pixel shader; all encapsulated by a hardware driver.

code scalable and portable across devices. The modular and flexible pipeline (Figure 3) is programmed in a custom shader language (similar to the well known GLSL or HLSL languages). We provide a linear-actuation focused SDSL, including a runtime environment and IDE with an interactive inspector and display simulator.

Pipeline

The SDSL *pipeline* (Figure 3) abstracts actuator placement, spatial configuration, and visual configuration. The pipeline consists of a series of programmable stages, each addressing a specific aspect of content generation. It adopts a "frame"-based approach: as new frames of input arrive (for either visual or physical output, or both), the pipeline executes the programmable stages and generates appropriate output.

A *hardware driver* is responsible for describing the hardware configuration to the pipeline: the actuator arrangement and its visual capabilities. The actuator arrangement is then processed by the *arrangement shader* and forwarded to the *actuation shader* which computes how the display will change it's shape. Finally, the *pixel shader* computes the visual output of the display. All shaders can have uniforms (custom parameters). One particularly important type of uniform is the *sampler* that provides access to external data sources in the form of an array of 1/2/3/4-dimensional vectors (e.g., a Kinect depth stream or touch input).

Hardware Driver

The driver encapsulates all hardware-specific details, making SDSL code portable across different shape-displays. It is implemented once per display and supports three main tasks:

1. **Provide the device profile,** specifying the visual resolution per actuator and optionally the maximal and minimal update speed, and the actuators' range of motion. These values configure the pipeline (number of pixel shader instances per actuator) and validate if the program can be correctly executed on the given display.
2. **Provide the actuator arrangement.** The physical arrangement of actuators can be dynamic; the hardware driver captures the instantaneous arrangement and provides it as input to the arrangement shader.
3. **Implement the computed height and color** by appropriately driving the actuators and visual display.

If the hardware supports features beyond linear actuation, the driver can supply additional information via custom uniforms. For example, if the actuators can sense their rotation, this data could be made available to SDSL programs as a 1D texture sampler (linear float array).

Applications and Runtime

SDSL programs seldom stand alone, but generally are part of a bigger application (much like graphics shaders). Applications can use the SDSL runtime to execute SDSL programs. The runtime supports real-time uniform configuration, so that applications can provide data to the pipeline. For example, to support interactivity, the encapsulating application can use the runtime to supply user input to a shader program.

Arrangement Shader

The optional *arrangement shader* transforms the physical actuator coordinates and normals into a more convenient

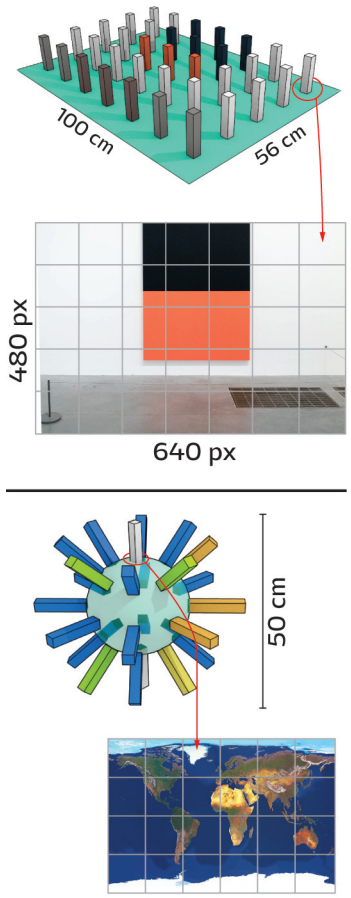

Figure 4: Different geometry shader applications. (left) mapping a physical actuator arrangement to the pixel space of a camera image. (right) Unwrapping a spherical actuator arrangement to a flat texture space.

coordinate space e.g., a low-density actuator grid defined in millimeters to a depth camera image defined in pixels (Figure 4, top). This shader can also be used to support non-planar actuator arrangements, e.g. unwrapping a sphere onto a planar texture (Figure 4, bottom).

Actuation Shader
This stage computes the physical state of each actuator. The computation is based on the actuator's location, normal vector, and custom application parameters. These shaders typically use external data sources through samplers. Our implementation currently assumes a linear actuation model (see Discussion on how this can be extended) that outputs a single value: the target height for each actuator.

Pixel Shader
The pixel shader computes visual output on a per-actuator basis. The displayed colors can be based on predefined imagery, mathematical models, a sampler or computed in real-time.

The visual resolution of shape-changing displays is often higher than their physical resolution (e.g. high-resolution displays attached to actuators [1], or projection on the actuated surface [5]). Pixel shaders support this resolution mismatch. For example, if a projection overlay is used, the hardware driver would specify that each actuator has a projected surface of $N \times M$ px. The pixel shader computes an RGBA (red, green, blue, alpha) color for each pixel within an actuator, based on the actuator's coordinates in the overall arrangement and the pixels coordinates within the actuator. This enables developers to apply complex sub-actuator detail across the entire display.

Language
SDSL is similar to existing shader languages [6, 9]. It follows the well known C-style syntax (including custom functions and block-scoping), and supports many "intrinsic" instructions familiar to graphics programmers. We support native vector and matrix arithmetic, as well as uniform declarations and samplers (externally provided arrays of 1/2/3/4-dimensional vectors). The language is type-safe and supports basic polymorphic functions.

SDSL code compiles to an intermediary representation, called *Shape-Display Shader Execution code* (SDSE). SDSE is completely linked, has all references resolved, and types checked. Using SDSE we can introduce compile-time optimizations and static program analysis. The latter could be used to check if a program can be displayed correctly given a certain device profile.

Integrated Development Environment (IDE)
To ease the development of shape-changing applications we provide a web-based Integrated Development Environment (IDE) for SDSL (Figure 1). This IDE is structured around an in-browser 3D simulation of the shape-changing display, enabling developers to test their programs without running the risk of damaging experimental hardware.

The syntax-highlighting code-editors (Figure 1, left) highlight syntax errors. The input and output of each shader stage can be inspected in real-time to ease debugging. Runtime errors and uniform property modifiers (Figure 1, right) allow the developer to experiment with parameters. All uniforms can be manipulated in real-time and texture sampler data can be provided using WebSockets. The arrangement and device profile can be configured within the IDE or provided via WebSockets.

PIXEL

```
uniform sampler3D colorData;
void main() {
  sd_PixelColor = vec4(colorData[iFragCoord.z] / 255.0, 1.0
}
```

ACTUATION

```
uniform sampler1D depthData;
void main() {
  sd_FragHeight = depthData[iFragCoord.z] / 255.0;
}
```

ARRANGEMENT

```
uniform vec2 dataRes;
void main() {
  vec2 inUnitySpace = iFragCoord.xy / iResolution.xy;
  vec2 coord = mat2(dataRes.x, 0.0, 0.0, dataRes.y) * inUnitySpace;
  float idx = floor(coord.x + coord.y * 640);
  sd_FragCoord = vec3(coord, idx);
}
```

Figure 5: Example Application: using depth data. The geometry shader maps the actuator arrangement to the Kinect space, the actuation shader uses the depth values and the pixel shader maps the image colors.

Implementation

SDSL is implemented in Ruby, including the IDE, using the Opal Ruby-to-Javascript compiler. SDSL code is parsed using the Treetop parser generator, compiled to SDSE using custom code, and uses the Ruby built-in string expansion to turn SDSE into Ruby code. A runtime environment, again written in Ruby, supports the execution of the SDSL pipeline. Using Opal, we can use this infrastructure in a web-browser.

Example Application: Depth Data as Input

Depth data, as provided by a depth camera (e.g., the Microsoft Kinect) or as computed from the height profile of terrain, is often used as shape display input. This depth data has to be mapped to the individual actuators. SDSL makes this task straightforward: we implement an arrangement shader that maps the physical hardware arrangement, first to the unity, then to the proper resolution space (Figure 4, left). The arrangement and pixel shader then sample the depth (and color stream) and transform it into actuator height and pixel color.

Discussion

We presented a first step towards hardware independent shape-changing displays. In the following we discuss how the concept of SDSL can be generalized and what its limitations are.

Supporting User-Input/Interactivity

The SDSL pipeline supports user-input at every stage through uniforms. For example, touch input on the display surface could be provided as 3D vector of the touch coordinates in space. More complex interaction is supported through samplers as demonstrated by the "Depth Data as Input" example.

Non-linear actuation models

Our pipeline can be adapted to other types of shape-change by extending the shader stages to output custom data structures, rather than pre-defined data types. The computer graphics community already uses such an approach, but applying it generically to shape-changing displays requires greater understanding of the requirements. If a display differs significantly from the linear actuation model we currently focus on, one could alter the pipeline itself. We implemented SDSL so that it's easy to implement custom pipelines. By open-sourcing SDSL, we hope to encourage the community to contribute new shaders that fulfill their needs.

Adapting to Hardware

Hardware is abstracted via drivers which provide the actuator arrangement and receive the computed actuator and pixel data. How the driver bridges between the hardware and pipeline, depends on the hardware itself and the runtime environment used to execute the SDSL code. Our Ruby based runtime environment supports TCP and WebSocket based I/O, so that the hardware driver is language agnostic.

Performance

Our SDSL implementation is able to drive all tested applications and shape-changing hardware in real-time. Pipeline execution time will increase depending on the computational complexity of the programs for each shader stage and linearly with spatial and visual resolution. While our Ruby-based runtime implementation trades performance for flexibility, the compiler was designed with extensibility in mind, so that the runtime can be implemented in a better performing language.

Limitations
As in graphics, there are scenarios ill-suited to a shader-based development model, e.g. user-input validation. Shape-display programming scenarios however, are centered around the parallel control of many actuators and thus well suited to a shader-based approach. Developing in a fixed programming model comes at the cost of flexibility. SDSL enforces fixed pipeline stages to enable cross-hardware interoperability. The pipeline can be changed at the cost of hardware interoperability.

Conclusion
Research into shape-changing displays continues to investigate new materials and hardware; driving the need for hardware-independent programming models. We present SDSL as an initial approach—inspired by mature graphics pipeline concepts—to foster discussion whilst simplifying shape-change motion design. We hope that our open-source implementation, including a simulator and IDE, will allow the community to share applications and inspire new ways of thinking about programming shape-changing displays.

References
[1] Alexander, J., Lucero, A., and Subramanian, S. Tilt Displays: Designing Display Surfaces with Multi-axis Tilting and Actuation. MobileHCI '12, ACM (2012), 161–170.

[2] Coelho, M., and Zigelbaum, J. Shape-Changing Interfaces. *Personal and Ubiquitous Computing 15*, 2 (2011), 161–173.

[3] Cook, R. L. Shade trees. SIGGRAPH '84, ACM (New York, NY, USA, 1984), 223–231.

[4] Follmer, S., Leithinger, D., Olwal, A., Cheng, N., and Ishii, H. Jamming User Interfaces: Programmable Particle Stiffness and Sensing for Malleable and Shape-changing Devices. UIST '12, ACM (2012), 519–528.

[5] Follmer, S., Leithinger, D., Olwal, A., Hogge, A., and Ishii, H. inFORM: Dynamic Physical Affordances and Constraints Through Shape and Object Actuation. UIST '13, ACM (2013), 417–426.

[6] Gray, K., and Corporation, M. *Microsoft DirectX 9 programmable graphics pipeline*. Developer Series. Microsoft Press, 2003.

[7] Poupyrev, I., Nashida, T., and Okabe, M. Actuation and tangible user interfaces: The vaucanson duck, robots, and shape displays. TEI '07, ACM (2007), 205–212.

[8] Rasmussen, M. K., Pedersen, E. W., Petersen, M. G., and Hornbæk, K. Shape-Changing Interfaces: A Review of the Design Space and Open Research Questions. CHI '12, ACM (2012), 735–744.

[9] Rost, R., and Kessenich, J. *OpenGL Shading Language*. Graphics programming. Addison-Wesley, 2006.

[10] Roudaut, A., Karnik, A., Löchtefeld, M., and Subramanian, S. Morphees: Toward High "Shape Resolution" in Self-actuated Flexible Mobile Devices. CHI '13, ACM (2013), 593–602.

[11] Vidimče, K., Wang, S.-P., Ragan-Kelley, J., and Matusik, W. Openfab: A programmable pipeline for multi-material fabrication. *ACM Trans. Graph. 32*, 4 (July 2013), 136:1–136:12.