

Towards a Computer Vision Shader Language

Gregor Miller, Steve Oldridge and Sidney Fels*
Human Communication Technologies Laboratory
University of British Columbia

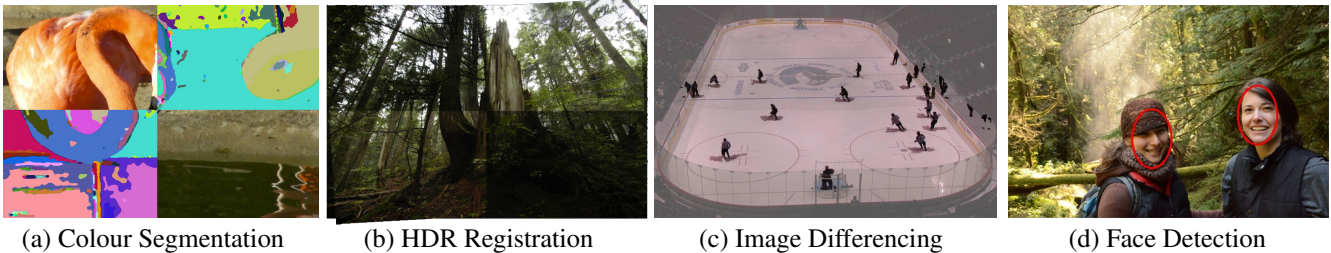


Figure 1: The Open Vision Language (OpenVL) is a new language analogous to a shader, designed to provide high-level access to computer vision methods covering a wide range of problems. OpenVL supports many vision problems; we have implemented segmentation (colour in (a)), various types of registration (such as HDR stacking in (b)), image differencing in (c) and face detection in (d).

1 Open Vision Language

Computer vision is a complex field which can be challenging for those outside the research community to apply in the real world. The problem preventing widespread adoption is the lack of a formulation which separates the understanding of a task from the knowledge of which algorithm to use as a solution. Successful abstractions in computer graphics, such as OpenGL and DirectX, hide the algorithmic details of rendering behind a powerful interface. We propose a similar abstraction for computer vision, analogous to a shader language, which provides developers with access to sophisticated vision methods without requiring specialist knowledge. Our contribution is an interface which presents developers with mechanisms to describe their vision task; the description is interpreted to select an appropriate method and provide a solution.

Many computer vision problems can be divided up into smaller sub-problems and solved by providing solutions to each sub-problem. This applies conceptually as well as algorithmically and so we base our vision shader language on this principle. We allow the user to describe vision tasks by dividing the conceptual problem into sub-tasks, then the sequence of sub-tasks is analysed to select a suitable method to apply. Our language is made up of these sub-tasks, which we term *operations*, and we provide a core set to span the range of computer vision problems.

Each of these operations accepts descriptions of the conditions under which they must operate. For example, if requesting detection we provide the developer with a means to describe the object to detect (e.g. a ‘face’, mostly front-facing, possible occlusion, multiple instances, as shown in Figure 1d) and the means to describe the image (detailed, varied illumination). From this description we can infer which face-detection algorithm will work most effectively under these conditions. Behind the scenes, each algorithm which is incorporated into our framework is evaluated with respect to how it performs under the conditions descriptions; this allows us to select the best one when the developer describes their problem.

The power of the abstraction comes from the evaluation of the operations as a sequence instead of individually; this approach allows us to establish the higher-level problem to solve and possibly select a single method capable of solving it more effectively.

2 Example Problems

For the first version of our shader language we support a small set of operations: `segment`, `match`, `detect`, `select` and `solve`.

We use `segment` as a means for the user to describe the image: conceptually it produces a segmentation, which becomes the mental model for the user (every operation applies to *segments*); if this is the only operation issued it will produce an actual segmentation, such as the one shown in Figure 1a. The type of segmentation depends on the description supplied. If this is not the only operation (and so the problem is likely not segmentation), the `segment` operation provides a description of the image, and so contributes to the conditions of the problem.

For example, image registration involves finding similar regions in two images and optimizing for the global alignment. We can express this in our language as: `segment`, `match`, `solve`. `segment` provides a description of the images, `match` accepts a set of *variances* as input to describe the differences between the images (such as intensity, as in Figure 1b) and `solve` is constrained to be `global`. Conceptually this is finding segments in each image, matching them across images and then optimizing for the alignment. Internally the language is using a single method to accomplish the entire task.

Use of the `select` method allows us to describe more problems, such as image differencing. First we `segment` the images, then we find the correspondences between them using `match`, and finally `select` the segments in the second image which have no matches in the first. A result of this is shown in Figure 1c. Finally, using `detect` we can let the user provide a template to match against in the input images. The template could be a model, an example image or a simple description. In this case, we provide a method to describe faces, and the sequence is simply `segment`, `detect`.

We are continuously expanding the problems we can solve using this small set of operations, and enhancing the flexibility of the description each uses to simplify access to computer vision for developers. We are also working on other operations to extend our work into multi-view and 3D, such as `project`, `intersect` and `calibrate`. We hope that the research into this methodology will provide simpler and more intuitive access to sophisticated computer vision methods for developers, hobbyists and researchers in other fields.

*e-mail: {gregor,steveo,ssfels}@ece.ubc.ca