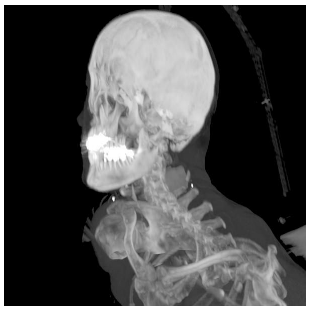
Shadie: A Domain-Specific Language for Volume Visualization

Category: System



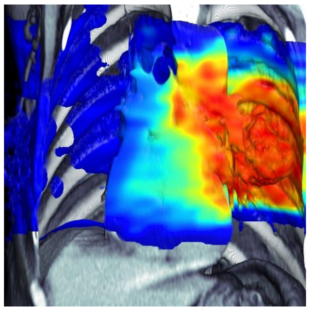
```
data = data3d('data/ct')

num_steps = length(E - S) / 0.002
m = 0.0

for t in linspace(0.0, 1.0, num_steps):
    # compute position along ray
    P = (1-t) * S + t * E

    # update maximum
    m = max(m, cubic_query_3d(data, P))

return m
```



```
# apply transfer function
tf_query = (density - tf_pos + tf_width)
    / (2 * tf_width)
if tf_query < 0: continue
rgba = linear_query_1d_rgba(tf, tf_query)
color = rgba.xyz

# compute colormapped dose
dose_value = linear_query_4d(dose, P, frame)
    * 32768
if dose_value > dose_threshold:
    color = linear_query_1d_rgba(colormap,
        dose_value / dose_max).xyz
```

Fig. 1. Left: A CT-scan displayed using a maximum-intensity projection, together with its complete Shadie source code. Right: A time-varying lung tumor visualization combined with a radiation dose displayed using a colormap, and a cut plane to view the inside of the lung. Part of the shader shown; the complete shader is 35 lines of code.

Abstract— Despite abundant research on volume rendering, its usage among medical professionals is still limited. We believe this is partly due to limited flexibility of existing systems, and the difficulty of customizing them without low-level software engineering expertise. In this paper we introduce Shadie, a GPU-based volume visualization framework built around the concept of shaders: self-contained descriptions of the desired visualization written in a high-level Python-like language that can be written by non-experts, often in 10-20 lines of code. Type inference and source-to-source translation to efficient CUDA code allow for interactive framerates. The concepts of rays, transfer functions or lighting are unknown to the core renderer; instead, they are purely features of the shaders. Furthermore, any number of datasets can be queried within a shader and combined in arbitrary ways. We show several examples from radiation oncology, where we combine time-varying CTs, radiation dose distributions, colormaps and transfer functions in custom visualizations. Similarly to RenderMan shaders commonly written by artists, our vision is that Shadie can be learned and used by medical physicists or other scientists to produce highly customized but interactive visualizations.

Index Terms—Volume visualization, GPU, biomedical imaging.

1 Introduction

Volume visualization has become invaluable in a wide variety of applications in medicine, engineering, and the sciences. Examples include visualization of 3D sampled medical data (CT, MRI), seismic data from oil and gas exploration, or computed finite element models. However, despite a large amount of research on volume rendering and the availability of interactive implementations, its usage among scientists and medical professionals is still limited [18, 19]. We believe this is due to limited flexibility of existing systems, and the difficulty of customizing them without low-level software engineering expertise. This situation is made worse with the emergence of heterogeneous systems with multi-core CPUs and many-core GPUs, requiring parallel programming experience.

While off-the-shelf visualization solutions such as Amira or OsiriX are successful, they offer limited customizability and force the developer into a rigid plugin framework. On the other hand, software toolkits such as vtk offer flexibility but are primarily suited for software engineers. There is currently a noticeable gap between low-level toolkits and all-in-one systems. We created Shadie to fill this gap and to make custom volume visualization more accessible to researchers and domain scientists with limited programming background. Our goal was to provides enough low-level control so users can perform custom mathematical operations when exploring and visualizing data, while focusing on the high level problems using a language that is concise and easy to learn. This is similar to the situation in computer graphics, where artists are developing custom shaders without having to worry about all the complexities of game engines or high-end rendering systems such as Renderan.

Our framework is built around the concept of shaders: code snippets in a high-level Python-like language that define the visualization and can be written by non-experts, often in 10-20 lines of code. While shaders written in languages like Cg and GLSL have long been used in volume renderers, these were usually hidden from the user and tightly coupled with complex C++ code of the renderer itself. In contrast, in Shadie the shader is a single, self-contained description of the desired visualization, and is passed to the renderer as an argument, freeing the user from issues like compilation, buffer management, CPU-GPU intercommunication, or data interpolation. Type inference and source-to-source translation to NVIDIA's CUDA allow for interactive frame rates. The concepts of rays, transfer functions, lighting computations, or cut-planes are unknown to the core renderer; instead, these are purely features of the shaders. Furthermore, traditional volume rendering can be easily combined with ray-traced implicit surfaces or maximum-intensity projections within the same image. Any number of 1D, 2D, 3D or 4D (i.e. time-varying) datasets can be combined in arbitrary ways to produce the final visualization. To evaluate Shadie, we show several examples from radiation oncology, where we combine time-varying CTs, radiation dose distributions, colormaps and transfer functions in custom visualizations.

2 Previous work

Commercial systems: In addition to volume viewers that are integrated into PACS workstations or medical scanners, several volume visualization applications are commercially supported. Popular examples include Amira [2], Osirix [27], and Fovia [11]. While most of these provide some extensibility through plugins or APIs, they are by and large closed systems with limited flexibility. Another tool must be adopted if a desired calculation is not supported or if some desired parameters are not exposed.

Dataflow environments: Several visualization frameworks adopt a dataflow methodology, where applications are constructed by connecting computational elements (modules) to form a program (net-

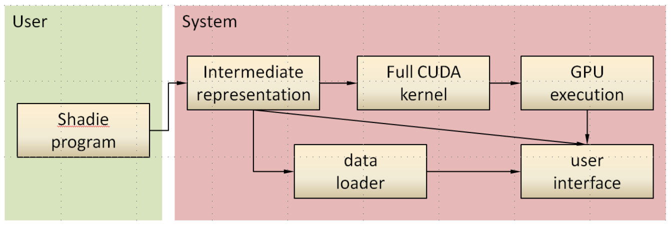


Fig. 2. An overview of our framework. The shader is all that the user has to write; the rest is handled by the system.

work). This includes AVS [36], IBM Data Explorer and OpenDX [1, 20], IRIS Explorer [9], SCIRun [29], VisTrails [6, 3], VolView [37], and vtk [32]. Despite the expressive power of these environments, the size and complexity of these systems may be a barrier to entry for new users. In addition, adding new functionality requires substantial programming background and incurs a steep learning curve.

Visualization calculators: The popularity of software packages such as MATLAB, Mathematica, Python/SciPy and R demonstrates the power of directly performing calculations while exploring and analyzing data. However, these systems are not optimized for large data and fail to provide high-quality interactive rendering. Moran and Henze [25] developed the Demand-Driven Visualizer to interactively specify derived fields and to expedite the computation of derived fields by using lazy evaluation. Similarly, Jankun-Kelly and Ma [17] describe a spreadsheet-like system for data exploration that provides an interface through a scripting language for performing operations on data. Both efforts do not make use of GPU acceleration and fail to be interactive for larger data sets. The Scout system by McCormick et al. [22] is a visualization DSL providing expression-based queries that are evaluated on the data. This system is perhaps the closest previous work to Shadie; the major difference is that Scout only calculates modified data which is then displayed by a fixed-function visualization algorithm, while our system completely defines the visualization itself.

Visualization toolkits: The availability of comprehensive open-source toolkits such as vtk [32], itk [16], the InfoVis Toolkit [10], Improvise [38], Prefuse [15], and Protovis [5] has vastly changed and simplified visualization practice and education. Most of these toolkits provide a collection of visualization modules that encapsulate visualization algorithms. They can be extended by creating new components from scratch or subclassing existing components. However, this requires significant software engineering effort and deep familiarity with the programming model.

Shading languages: Shading languages were first introduced with Shade Trees [8] and popularized with the RenderMan Shading Language (RSL) [14]. The PixelFlow system introduced the first real-time shading shading language [26], an approach that has been adopted by GPUs with shader languages such as Cg [21], HLSL [30], and GLSL [31]. McGuire et al. [23] propose a framework to automatically combine shaders written for GPUs. Several shader languages were developed for ray tracing and global illumination, such as the Vision global illumination system [34, 35], the BMRT ray tracer [13], and RTSL [28]. Shadie follows a similar philosophy, but focuses on volume visualization; furthermore, it provides a self-contained description of the whole visualization, which is not true about other shading languages (e.g. Cg requires complex binding to C code, while RenderMan needs a separate stream of commands).

3 Design Goals and Decisions

The language and system design is guided by a handful of high-level goals:

Expressibility: The system should be capable of combining multiple volumetric datasets, commonly available for a single patient. It should allow for the application of a number of different transfer functions or colormaps, and be capable of combining implicit surfaces with volume renderings. Full support for 4D datasets and time variation is also required. While some of this functionality could be added to existing frameworks, the changes required would be quite invasive.

Ease of adoption: The target users of the system are domain scientists like medical physicists, who are experienced with mathematics and with scientific programming tools like MATLAB and Python/SciPy, but not low-level software engineering. Therefore, we want to use a familiar, existing syntax and programming model, while striving for maximum simplicity of the code and exposing a number of domain-specific features for convenience. The system should also accept data in common formats used in medical imaging.

Performance and visual quality: We desire interactive performance, close to existing fixed-function volume visualization implementations. Language features that would significantly slow down rendering are therefore not acceptable. The rendering quality should also be comparable to fixed-function systems.

Compactness: A complete visualization mode together with default parameter settings should be defined within a single file, and running it should be as simple as passing this file to the system, with no additional compilation, linking or configuration file management. The complexity of the implementation itself should be kept to a minimum, enabling ease of future extension.

Hardware abstraction: The system should take maximum advantage of GPU acceleration, but at the same time should not be tightly constrained to a single model of GPU computation. Instead, a potential future switch from a single GPU to a GPU cluster, or to a hybrid CPU-GPU system, or from CUDA to OpenCL should require only a change in the back-end, thus being transparent to the user.

Based on these goals, we made some key design decisions:

Python-inspired DSL: We decided to embed the Shadie language into Python syntax, which leads directly to syntax familiarity for users of scientific programming environments, and satisfies the hardware abstraction requirement. Furthermore, we are able to take advantage of an existing Python parser. Python syntax is also sufficiently rich to allow for future extensions to Shadie.

Ray-tracing model: Evaluation of functions along rays is an easily understood concept, it is natural for volume visualization, allows for high visual quality, and is a good fit for data-parallel computation. In our design, a Shadie program is simply a function that takes a ray as input (specified by its two endpoints) and returns a color for the corresponding pixel. Furthermore, a time input to the function can be used to produce 4D (time-varying) visualizations. This model is flexible enough to satisfy our expressibility constraints, while still simple enough to be controlled by very few lines of code. A disadvantage of ray-tracing is that some rendering modes (e.g., splats and glyphs) do not map to it as naturally; designing an extension to Shadie to support these modes would be an interesting future direction.

A domain-specific library and type system: We provide domain-specific data types (floats, integers, 2-, 3-, and 4-dimensional vectors, volumes, images) and operations (e.g. linear and cubic interpolation, gradient evaluation, vector operations, reading common image and volume formats).

Currently Shadie only handles regular volumetric data and does not deal with curvilinear and irregular volumes or unstructured point data. The ray tracing model is flexible enough to adapt these data types, but they will require additional features in our language and library (e.g., mesh and point set data types, and operators for MLS interpolation). This is subject for future work.

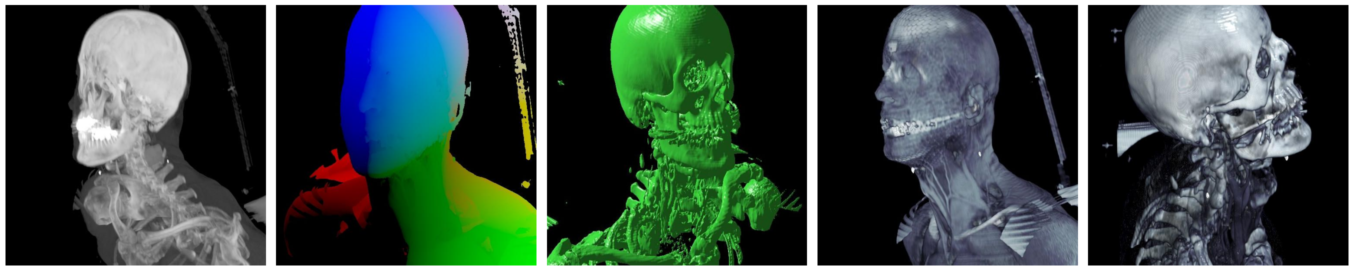

(a) MIP (b) Position shader (c) Implicit surface (d) Volume render (e) Volume + Phong

Fig. 3. Screenshots illustrating the shaders from Section 4.

4 Language Introduction by Examples

A Shadie program can be thought of as a sequence of statements that, given a ray segment, computes the color of the pixel that the ray passes through. All visualization is assumed to take place within the unit cube, $[0,1]^3$, and the system automatically clips rays to the cube, taking care of camera movement and transformation matrices. The program can access a number of predefined variables, the most important being `S` and `E`, the start and end of the ray. A typical program will march the ray in small steps, querying datasets and combining the queries into colors. (However, this is not required, and the program could ignore the ray and compute a completely unrelated quantity).

In addition, a Shadie program usually begins with a number of parameter definitions, where the parameters are either datasets (1D, 2D, 3D or 4D) or floating point constants (scalar or vector), which are bound to the GUI and modifiable on the fly. We now introduce the Shadie DSL using five examples with increasing complexity.

4.1 Maximum-intensity projection

In our first example, we show the complete Shadie implementation of the simple but commonly used *maximum-intensity projection* (MIP), where each pixel displays the maximum value of the volume encountered along the corresponding ray:

```
data = data3d('data/ct')

num_steps = length(E - S) / 0.002
m = 0.0

for t in linspace(0.0, 1.0, num_steps):
    # compute position along ray
    P = (1-t) * S + t * E

    # update maximum
    m = max(m, cubic_query_3d(data, P))

return m
```

The first line of the shader defines the dataset to load; here we use a CT-scan of a human head and neck. Next, the number of steps to take along the ray is adapted to the ray length. Ray-marching is performed using the convenient `linspace(a, b, n)` syntax, which uniformly distributes n samples between a and b (n does not need to be integer). The body of the loop simply computes the location along the ray and updates the current maximum with the value queried at that location using the `cubic_query_3d` function, which also provides cubic spline interpolation using the technique of Sigg and Handwiger [33].

Running this shader, stored in a file called `mip.py`, is as simple as typing:

```
shadie mip.py
```

This will bring up an interactive viewer, allowing the user to rotate and zoom; rendering single frames into EXR images is also supported by additional command-line arguments. The additional `-p` parameter will use 2×2 progressive sub-sampling to increase performance while camera is moved.

4.2 Position shader

In this example we ray-cast an implicit surface, displaying intersection points simply as RGB values. We define a threshold CT value, and march the ray until the threshold is crossed, using the current location as the intersection. (The root could be refined by bisection, but we omit this for simplicity.)

```
data = data3d('data/ct')

# parameters that can be modified within the GUI
thr = float_param(100, -200, 400, 'T')
step = float_param(0.002, 0.01, 0.005, 'S')

steps = length(E - S) / step

for t in linspace(0.0, 1.0, steps):
    P = (1-t) * S + t * E
    if cubic_query_3d(data, P) * 32768 > thr:
        return pos

return 0
```

Note that the parameters `thr` and `steps` have been defined using the `float_param` construct, which allows them to be modifiable by the user in the GUI or from the command-line. The arguments of `float_param` specify the default value, minimum, maximum and a key that can be used to select the parameter. The multiplication by 32768 is currently necessary to convert the value from a normalized range of $[-1,1]$ into the standard Hounsfield units of CT; this requirement should be removed in the future.

4.3 Implicit surface with Phong shading

Next, we extend the above implicit surface shader with phong lighting:

```
data = data3d('data/ct')
thr = float_param(100, -200, 400, 'T')
step = float_param(0.002, 0.01, 0.005, 'S')
lightpos = float3_param(10, -10, 10)
kd = float3_param(0.05, 0.74, 0.05)

steps = length(E - S) / step
C = normalize(S - E)

for t in linspace(0.0, 1.0, steps):
    P = (1-t) * S + t * E

    if cubic_query_3d(data, P) * 32768 > thr:
        N = -normalize(cubic_gradient_3d(data, P))
        L = normalize(lightpos - P)
        return phong(L, N, C, kd, 1, 20, 0.1)

return 0
```

The key operations are: computing C, the direction towards the camera, determining a normal vector from the gradient queried using `cubic_gradient_3d`, defining a lightsource direction, and calling the built-in `phong` function (but a custom shading model could also be applied easily).

4.4 Volume rendering with a transfer function

Implicit surfaces are not ideal for exploring CT data; instead, traditional volume rendering with a transfer function provides a higher-quality, cleaner visualization. The complete Shadie code looks as follows:

```
data = data3d('data/ct')
step = float_param(0.002, 0.01, 0.005, 'S')

# transfer function params
tf = data1d_rgba('bone.png')
tf_pos = float_param(150, 0, 200, 'P')
tf_width = float_param(100, 10, 400, 'W')

steps = length(E - S) / step
result = float3(0)
see_through = 1.0

for t in linspace(0.0, 1.0, steps):
    pos = (1-t) * S + t * E

    # query CT
    density = cubic_query_3d(data, pos) * 32768

    # apply transfer function
    tf_query = (density - tf_pos + tf_width)
        / (2 * tf_width)
    if tf_query < 0: continue
    rgba = linear_query_1d_rgba(tf, tf_query)

    # accumulation
    result += see_through * rgba.w * rgba.xyz
    see_through *= 1 - rgba.w

    # early ray termination
    if see_through < 0.01: break

return result
```

The transfer function is stored in a 1×256 RGBA image and queried using `linear_query_1d_rgba`, though one might as well compute a transfer function using just arithmetic operations. The variables `tf\_pos` and `tf\_width` control the location and width of the transfer function when applied to the CT data. Not that Shadie supports *swizzle operators* introduced in the Cg language using the syntax `rgba.xyz`, allowing for extraction of multiple components of a vector at once. (The above shader is slightly incorrect, because the transparency values returned from the transfer function should correspond to extinction over a fixed distance, and should be corrected if the step size changes; we ignore this for simplicity.)

4.5 Volume rendering with Phong shading

Finally, we apply Phong shading to the colors supplied by the transfer function, improving the perception of surface orientation and curvature. Furthermore, we also introduce the `opacity` variable to enable additional scaling of the alpha values returned from the transfer function. Below we present the modified loop:

```
for t in linspace(0.0, 1.0, steps):
    P = (1-t) * S + t * E

    # query CT
    density = cubic_query_3d(data, P) * 32768

    # apply transfer function
    tf_query = (density - tf_pos + tf_width)
        / (2 * tf_width)
    if tf_query < 0: continue
    rgba = linear_query_1d_rgba(tf, tf_query)
```

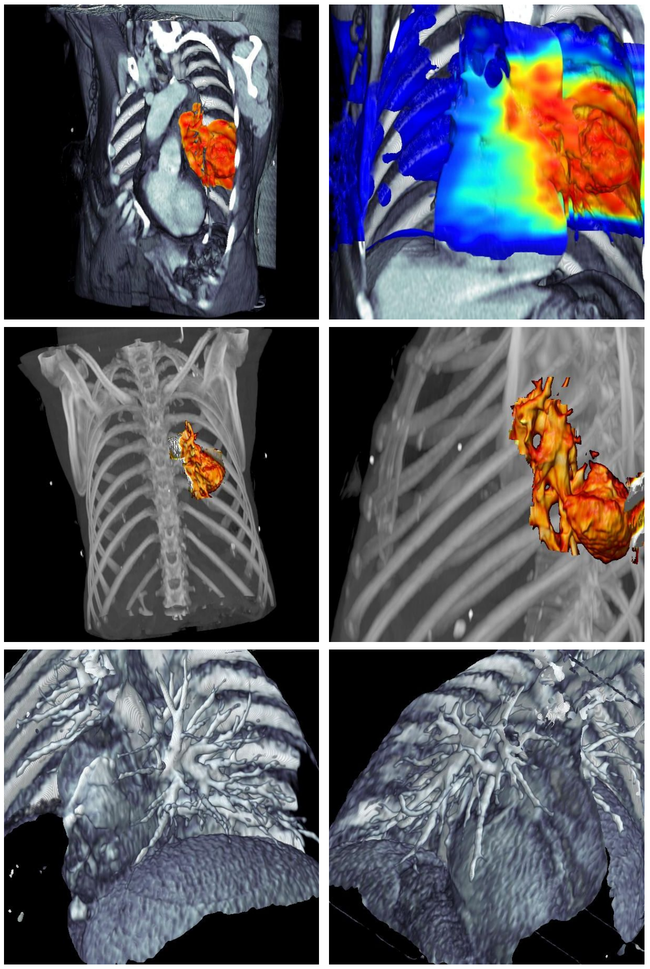


Fig. 4. Images from radiation oncology applications of Shadie, described in Section 5. Top: 4D (time-varying) visualization of a lung tumor using a cut-plane, with the radiation dose displayed as a colormap, showing two different dose thresholds. Middle: A combination of volume rendering within delineated tumor area, and maximum-intensity projection outside. Bottom: A view of the lungs computed by skipping the chest wall within the shader.

```
    # Phong shading
    N = -normalize(cubic_gradient_3d(data, P))
    L = normalize(lightpos - P)
    color = phong(L, N, C, rgba.xyz, 1, 50, 0.5)

    # accumulation
    result += see_through * rgba.w * opacity * color
    see_through *= 1 - rgba.w * tf_opacity

    # early ray termination
    if see_through < 0.01: break
```

In about 20 lines of Shadie code (not counting comments and blank lines), we implemented a fairly full-featured volume renderer and defined its default inputs and parameter values. In Section 5 we explore more complex examples.

5 Radiation Oncology Examples

Radiation oncology aims to eliminate tumor cells by precisely targeting them with large doses of X-ray or proton radiation. It is an area that can utilize an immense amount of volume data about any given patient – CT, MRI and PET scans, radiation dose distributions computed by Monte Carlo simulation, tumor delinations manually drawn by physicians, etc. Some or all of these datasets can be time-varying (four-dimensional), which becomes particularly important when targeting tumors affected by motion due to the breathing cycle. Therefore, we think that radiation oncology is a good example of a domain that could greatly benefit from Shadie's ability to produce interactive volume visualizations that combine multiple datasets in custom ways.

In the following, we show three examples inspired by radiation oncology, each of which can be expressed by short and elegant Shadie code, but would require relatively invasive changes to off-the-shelf volume renderers. The results of these visualizations can be seen in Figure 4 and also in the accompanying video.

5.1 Combined 4D anatomy and dose

This example shows a 4D (i.e. time-varying) CT anatomy of a patient with a lung tumor, together with a time-varying radiation dose distribution overlayed as a colormap. We also need a cut plane to be able to view the inside of lungs. To implement this in Shadie, we introduce a few extensions to the last volume rendering shader from Section 4. First, we use `data4d` to load the datasets:

```
ct = data4d('data/lung/ct')
dose = data4d('data/lung/dose')
```

We also load the colormap as a 1D image, define a threshold and maximum for the colormapped dose display, and a cut plane distance:

```
colormap = data1d_rgba('jet.png')
dose_threshold = float_param(100, 0, 200, 'T')
dose_max = float_param(196, 0, 200, 'M')
cut = float_param(-1, -1, 1, 'C')
```

To achieve time variation, we compute the frame number from the predefined variable `T`, which specifies the current time in seconds, and use the 4D versions of the query functions. Shadie also allows for specifying a cut plane normal and distance directly in the query function for convenience (though other cut volumes could be implemented directly within the shader):

```
# compute frame number
frame = fmod(T*5, 10)

# query 4D CT using a cut plane
density = cubic_query_4d_cut(ct, P, C, cut, frame)
        * 32768
```

Within the ray marching loop, we can then query the dose value, compare to threshold, and replace the color given by the transfer function by the one returned by the colormap if necessary:

```
# compute colormapped dose
dose_value = linear_query_4d(dose, P, frame) * 32768
if dose_value > dose_threshold:
        color = linear_query_1d_rgba(colormap,
                dose_value / dose_max).xyz

# use color for shading as usual ...
```

The result is shown in the top of Figure 4.

5.2 Combined volume and MIP rendering

Before radiation treatment, a physician usually draws the outline of the tumor on 2D slices. This delineation can be treated as another volume, where a value of 1 indicates a point inside the tumor volume and 0 outside. We show a visualization making use of this information, using a volume-rendered representation of the tumor and a maximum-intensity projection for rays that miss the tumor. This technique can be useful for locating the tumor, which can be difficult using only transfer functions and cut planes. The Shadie implementation of the idea is straightforward:

```
m = 0.0

for t in linspace(0.0, 1.0, steps):
    P = (1-t) * S + t * E

    # query CT
    density = cubic_query_3d(ct, P) * 32768

    # outside tumor volume?
    if linear_query_3d(itv, P) == 0:
        m = max(m, density)
        continue

    # apply transfer function as usual ...

if result == float3(0):
    return m * pow(2, exposure)

return result
```

The result is shown in the middle of Figure 4.

5.3 Unobstructed view of lungs

In this example, we inspect the bronchial tubes of a different patient by traditional volume rendering, but starting the accumulation only after the ray gets inside the lung. This is achieved by defining two thresholds, τ_1 and τ_2, and starting the rendering only after the CT density first crosses *above* τ_1 and then *below* τ_2, thus skipping the chest wall. To implement this in Shadie, we keep track of the *phase* the ray is in: phase 0 occurs before crossing τ_1, and similarly phase 1 lasts until τ_1 is crossed; phase 2 then triggers volume rendering:

```
phase = 0

for t in linspace(0.0, 1.0, steps):
    P = (1-t) * S + t * E

    density = cubic_query_3d(data, P) * 32768

    if phase == 0 and density > tau1: phase = 1
    if phase == 1 and density < tau2: phase = 2

    if phase == 2:
        # continue with standard volume rendering
```

The result is shown in the bottom of Figure 4.

5.4 Radiotherapy-specific computation within the shader

In the final example, we apply the system to computation and visualization of quantities specific to radiation oncology directly in the shader. For any given point of the rendered anatomy, we compute the standard deviation of the radiation dose (treated as a time sequence), which helps in identifying areas of large variation due to lung movement and thus an increased danger of incorrect treatment. With a similar approach, we can estimate the probability of tumor cell survival given the dose distribution using the linear-quadratic model from radiobiology [12]. The results are shown in Figure 5.

6 IMPLEMENTATION

In this section, we describe the implementation of Shadie: type inference, translation to intermediate representation, CUDA kernel generation, and rendering. Note, however, that this does not correspond chronologically to how we developed Shadie; in reality, we implemented the system by writing a fixed-function volume renderer in CUDA and then *taking away* functionality from the renderer, making it accessible only through shaders, and finally raising the level of abstraction of the shaders.

6.1 Type system and inference

Since CUDA is an explicitly typed language, we have to determine the type of every variable in a Shadie program at translation time. This would be impossible for general Python code, where the type of a variable can depend on the actual execution path, and therefore cannot be determined at compile-time; an overview of the issues can be found in [7]. We also considered a Hindley-Milner type system [24],

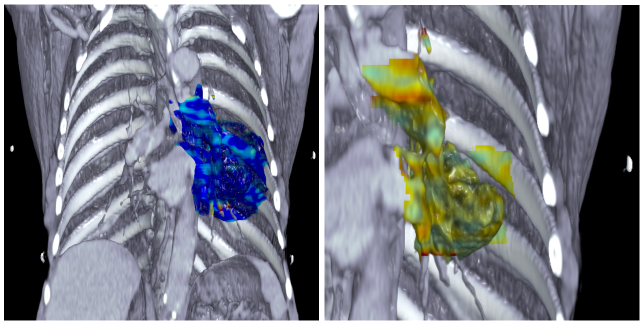


Fig. 5. Left: Standard deviation of the radiation dose treated as a time sequence, computed directly in the shader and displayed for points above a given dose threshold. Right: the probability of tumor cell survival aftr 5 radiation fractions, estimated using the linear-quadratic model.

where the most general type is determined for each variable by solving a system of type equations; however, we found that we would have to extend the system with a fixed hierarchy of numeric type classes similar to the one in the NESL language [4]. The complexity of such a system would not be suitable for a DSL designed for non-experts.

Instead, we designed a simple type system for Shadie, where every variable is of type `data`, `boolean`, `int`, `float`, or a 2-, 3-, or 4-element vector of `int`-s or `float`-s, e.g. `float3` or `int2`. (We also intend to add suppport for 2×2, 3×3 and 4×4 matrices of `float`-s in the future.) The language supports coercion (automatic typecasting) between several pairs of types, e.g., an `int` can be coerced to a `float`, a `float` can become a `float3`, etc. We can denote this relationship by the $\sqsubseteq$ relation, as in `float` $\sqsubseteq$ `float3`. Figure 6 gives a graphical overview of the type system.

Furthermore, Shadie distinguishes between *functions* and *operators*. Functions have a single type signature, e.g., the function `length` always takes a single `float3` argument and returns a single `float` (though functions will automatically coerce their arguments, so `length(1)` will be accepted as valid and return $\sqrt{3}$). On the other hand, operators accept exactly two arguments of any *comparable* types, i.e., where one of the types can be coerced to the other, and return the higher of the two types (with respect to the $\sqsubseteq$ relation). Therefore, an expression that adds, say, an `int` and a `float3` is acceptable and will have the obvious effect of adding the `int` to all three elements of the `float3`. There are some paradoxical cases: for example the "functions" `min` and `max` are in fact operators, while inequality relations (but not equality) are actually functions, and cannot be used ot compare vectors.

To enable efficient type inference, we introduce the following rules, which differ from standard Python semantics:

1. Every variable's type is determined at its first assignment, and cannot change later.
2. A variable defined in a nested statement block is local to that statement block.

The first rule guards against cases where a variable determined to be `float` is later assigned an `int2` value, or similar. The second rule protects against situations where one branch of an **if**-statement defines a variable to be `float` while the other branch makes it a `boolean`. The two rules also allow for a straightforward implementation of type inference in a single pass. We implement the type inference and translation in the Haskell programming language, which is ideally suited for source code manipulations.

6.2 Translation and execution

After types of all variables have been determined, we translate the shader into an intermediate representation, which constitutes a valid

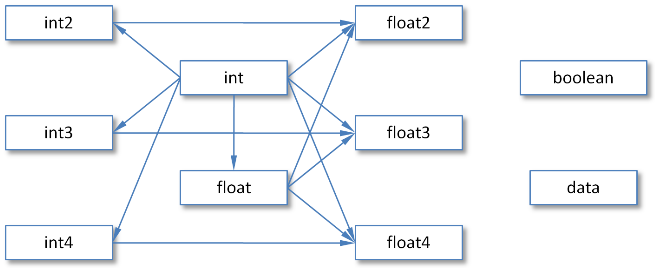


Fig. 6. The simple type system of Shadie. The directed edges in the graph correspond to types where automatic coercion is supported.

CUDA code snippet (though not a complete kernel). For example, the MIP shader from Section 4 will be translated into:

```
DATA3D(data, 'data/ct')
float num_steps = length(E - S) / 2.0e-3;
float m = 0.0;
float t = 0.0;
for (; t < 1.0; t = t + (1.0 - 0.0) / (num_steps - 1))
{
    float3 P = make_float3(1 - t) * S
        + make_float3(t) * E;
    m = max(m, cubic_query_3d(data, P));
}
return make_float3(m);
```

Note that type declarations have been filled in, and some explicit typecasts have been inserted; the `linspace` operation was expanded to a standard C-style loop. Furthermore, the `data3d` defnition has been translated to a special `DATA3D` directive; this is also the case for any parameter definitions.

An advanced user can write shaders directly in the intermediate language; passing a `.cu` file to Shadie works just as well as a `.py` file. This can be useful if advanced features of CUDA are required (e.g. pointers or shared memory).

The intermediate representation is converted into a full CUDA kernel in several steps:

- The dataset definitions are converted into CUDA texture declarations. Furthermore, the data is loaded into GPU memory and bound to the textures. The system currently supports the DICOM and MHA formats, common in medical imaging, plus a number of image formats.
- The parameter definitions are parsed and used to create a custom `Params` structure, which is filled and sent to the kernel for every rendered frame (since the parameters can be changed on the fly).
- A header and footer is added to the kernel. The header contains a number of convenience functions, including many versions of linear and cubic queries, while the footer defines the actual entry point and takes care of matrix transformations, ray clipping to unit cube, etc.

Finally, the CUDA kernel is compiled, and executed for every frame rendered.

7 Performance

Our video sequences were recorded on a single workstation with 4 quad-core Intel i7 CPUs and a single NVIDIA Tesla C1060 used for rendering; the CPUs are currently largely unused. A GeForce 260 was used as a display adapter.

In most of our examples, we use 2×2 subsampling while the user interacts with the system, and refine to a single ray per pixel when interaction stops, thus increasing the interactive performance about 4 times at the expense of temporarily lower quality; this feature is not

Example	frame time (s)	fps
MIP	0.24	4.2
Position	0.17	5.9
Implicit	0.13	7.7
TF	0.16	6.3
Phong	0.17	5.9
Dose	0.25	4.0
MIP + Vol	0.35	2.9
Lung	0.14	7.2

Table 1. The median frame rendering times for our examples, and the corresponing median frames-per-second.

used for the time-varying view. Table 1 shows rendering times per frame and frames per second, computed as the median difference between timestamps of successive frames in our recordings. Note that cubic interpolation for queries and gradients is the most expensive operation in our shaders; linear interpolation or larger step size can be used to speed up rendering at the expense of lower quality.

8 Conclusions and Future Work

We present Shadie, a domain-specific language and framework for rapid development of complex custom volume visualizations. The framework is designed to be simple enough to be used by non-programmers, while still providing interactive performance. We have demonstrated the effectiveness of the system in several examples from radiation oncology. We see a lot of potential in combining computation and visualization in our framework, and we believe domain-specific languages are the right tool to expose the massive parallelism of GPUs to non-expert users.

An interesting future direction would be to augment the language with more data types, such as irregular meshes and unstructured point clouds. These data types will require additional operators, such as MLS interpolation. Language features like user-definable functions and records would be very useful and relatively easy to implement. We also want to investigate if preintegration can be added as a language feature, potentially improving rendering quality and performance at the expense of a precomputation step. We also plan to expand the automatically generated GUI and enable others to implement custom GUI visualization tools on top of Shadie. Finally, we are planning to make the language and framework openly available and hope to get a user community to contribute custom shaders.

References

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *IEEE Visualization*, pages 263–270, 1995.

[2] Amira. http://www.amira.com/.

[3] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, and H. T. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization*, pages 135–142, 2005.

[4] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *SIGPLAN Not.*, 28(7):102–111, 1993.

[5] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121 – 1128, 2009.

[6] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: visualization meets data management. In *ACM SIGMOD international conference on Management of data*, pages 745–747, New York, NY, USA, 2006. ACM.

[7] B. Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University San Luis Obispo, 2005.

[8] R. L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH)*, pages 223–231. ACM Press, 1984.

[9] I. Explorer. http://www.nag.co.uk/welcome_iec.asp.

[10] J.-D. Fekete. The infovis toolkit. In *IEEE InfoVis*, pages 167–174, 2004.

[11] Fovia. http://www.fovia.com/.

[12] J. F. Fowler. The linear-quadratic formula and progress in fractionated radiotherapy. *Br. J. Radiol.*, 62:679–694, 1989.

[13] L. Gritz and J. K. Hahn. BMRT: A global illumination implementation of the RenderMan standard. *J. Graph. Tools*, 1(3):29–48, 1996.

[14] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH)*, pages 289–298. ACM Press, 1990.

[15] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proceedings of ACM CHI*, pages 421–430, 2005.

[16] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide*. Kitware Inc., 2005.

[17] T. J. Jankun-Kelly and K.-L. Ma. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, 2001.

[18] C. Johnson, R. Moorhead, T. Munzner, H. Pfister, and P. Rheingans. Visualization research challenges. Technical report, NIH/NSF, Jan 2006.

[19] B. Lorensen. The state of medical visualization: A personal view. Presentation at NIH/NSF VRC workshop, Sept 2004.

[20] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An architecture for a scientific visualization system. In *IEEE Visualization*, pages 107–114, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[21] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH)*, pages 896–907, 2003.

[22] P. McCormick, J. Inman, and J. Ahrens. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *IEEE VIsualization*, pages 171–178, Jan 2004.

[23] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi. Abstract shade trees. In *Symposium on Interactive 3D Graphics and Games*, pages 79–86, March 2006.

[24] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[25] P. Moran and C. Henze. Large field visualization with demand-driven calculation. *VIS '99: Proceedings of the conference on Visualization '99: celebrating ten years*, Oct 1999.

[26] M. Olano and A. Lastra. A shading language on graphics hardware: The Pixelflow shading system. In *Computer Graphics (Proceedings of SIGGRAPH)*, pages 159–168. ACM Press, 1998.

[27] OsiriX. http://www.osirix-viewer.com/.

[28] S. Parker, S. Boulos, J. Bigler, and A. Robison. RTSL: A ray tracing shading language. *IEEE Symposium on Interactive Ray Tracing*, Jan 2007.

[29] S. Parker, D. Weinstein, and C. Johnson. *The SCIRun computational steering software system*. Birkhauser, 1997.

[30] C. Peeper and J. Mitchell. Introduction to the directx 9 high level shading language, 2003.

[31] R. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2004.

[32] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice-Hall Inc., 2nd edition, 1997.

[33] C. Sigg and M. Hadwiger. *Fast Third-Order Texture Filtering*, pages 313–329. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, 2005.

[34] P. Slusallek, T. Pflaum, and H. Seidel. Implementing RenderMan - Practice, problems and enhancements. *Computer Graphics Forum*, 13(3):443–454, 1994.

[35] P. Slusallek, T. Pflaum, and H. Seidel. Using procedural RenderMan shaders for global illurnination. *Computer Graphics Forum*, 14(3):311–324, 1995.

[36] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics & Applications*, 9(4):30–42, 1989.

[37] VolView. http://www.kitware.com/products/volview.html.

[38] C. E. Weaver. Building highly-coordinated visualizations in improvise. In *IEEE InfoVis*, pages 159–166, 2004.