



Property-Based Testing for Visualization Development

M. Stegmaier^{1,*}, D. Engel^{2,*} , J. Olbrich¹, T. Ropinski², M. Tichy¹ 

¹Ulm University, Institute of Software Engineering and Programming Languages, Germany

²Ulm University, Institute of Media Informatics, Germany

*equal contribution

Abstract

As the testing capabilities of current visualization software fail to cover a large space of rendering parameters, we propose to use property-based testing to automatically generate a large set of tests with different parameter sets. By comparing the resulting renderings for pairs of different parameters, we can verify certain effects to be expected in the rendering upon change of a specific parameter. This allows for testing visualization algorithms with a large coverage of rendering parameters. Our proposed approach can also be used in a test-driven manner, meaning the tests can be defined alongside the actual algorithm. Lastly, we show that by integrating the proposed concepts into the existing regression testing pipeline of Inwiwo, we can execute the property-based testing process in a continuous integration setup. To demonstrate our approach, we describe use cases where property-based testing can help to find errors during visualization development.

CCS Concepts

• **Human-centered computing** → **Visualization toolkits**; • **Software and its engineering** → **Software verification and validation**;

1. Introduction

Testing plays a crucial role in ensuring software quality. When performed manually, it is labor-intensive and time-consuming. For this reason, it seems natural to automate testing as much as possible. Automated testing not only saves time, but also contributes to reproducible tests. In addition, the lower cost of automated testing compared to manual testing allows testing to be more thorough. In the context of test-driven development [Bec03], software tests are written in advance. A software requirement first gets converted to a test and only afterwards or in parallel the implementation of the requirement is developed.

While the automatic generation of tests is often straight-forward in the case of unit tests, it is significantly more difficult to test the interaction of different modules. This process is called *feature interaction testing* [PSK*10] (FIT). One of the difficulties that arise with FIT, is that through the many possibilities to combine modules for interaction, the resulting space of possible tests results in a combinatorial explosion. However, studies on debugging with feature interaction testing have shown that while testing the interaction of features in pairs already significantly increases the error detection rate, the increase from combining features in pairs to combining three or more features is insignificant [WK02, KWG04]. Thus, with pairwise coverage, a good balance between error detection rate and scalability can be obtained.

Data visualization techniques are used in a wide variety of research fields, such as medicine [KMM*18], biology [RE07], bio-

chemistry [KKF*17] and physics [AAA*19]. As data visualization demands increase, and researchers as well as practitioners rely on visualizations to make critical decisions [KOJC13], it is of utmost importance to verify visualizations for their correctness. This includes software quality assurance for the development of the rendering algorithms underlying visualizations. While verification and rigorous testing is applied in a wide range of disciplines in computer science, the adoption of those processes has been rather slow in visualization [KS08]. This lack of rigorous software testing often prevents researchers from sharing their visualization prototypes, negatively impacting the reproducibility and practicality of their work. We believe the reason for this is that visualization algorithms are difficult to test due to their large set of parameters and their often hard to declare impact on the resulting rendering.

In this work, we focus specifically on the verification of the code implementations underlying visualization algorithms. Unfortunately, very little research has been conducted on appropriate testing concepts for visualization software. Current visualization software commonly makes use of unit tests and simple regression tests. We argue that those types of tests alone are insufficient to reliably verify the correctness of visualization implementations, as unit tests fail to capture errors occurring through the interaction of different software components. Current frameworks try to identify those integration errors using simple regression tests, where the resulting image of a visualization is compared to a reference image that was produced when the test was created. The problem with that

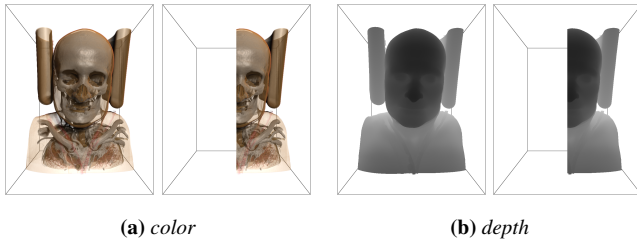


Figure 1: An example of volume clipping. We can test clipping implementations by verifying that the amount of background pixels increases, when increasing a lower bound clipping plane. The bounding box denotes the original volume borders.

approach is that the visualization is only tested with the parameter configuration that was used to create the reference image. However, complex visualization algorithms have many different parameters and the current testing methodology may fail to find errors that only occur with a different configuration. Furthermore, different graphics hardware and driver versions have been reported to produce slightly deviating results, which introduces issues, when the system that is running the tests is changed.

To alleviate these issues we propose to use the concept of property-based testing [FB97] for visualization development. Property-based testing allows for defining certain expectations in the way a rendering changes, given a certain change in the rendering parameters. For example, we can test the implementation of volume clipping. Volume clipping is the process of cutting a volumetric data set using axis-aligned planes, usually to get insights of the interior or to hide irrelevant information. Figure 1 shows volume clipping by increasing the lower bound clipping plane along the X axis. To test the correctness of a clipping implementation, we could for example check if shifting the clipping bound along an axis actually increases the amount of background pixels visible. Due to its illustrative nature, we use this example throughout the paper to explain the concepts behind and the implementation of our approach.

Our implementation is based on Inviwo [JSS*19], an extensible visualization framework that allows to build visualization pipelines in a graphical user interface. Users can directly model the data flow through a network of functional units, analogous to the visualization pipeline. We give a detailed introduction to the relevant parts of Inviwo in Section 3.2, including its current testing capabilities. Our implementation of property-based tests in Inviwo is also compatible with its current regression test system, and thus contributes to more reliable implementations within Inviwo. Note that our efforts to cover larger parts of Inviwo with property-based testing are still ongoing, therefore the focus of this work lies on the underlying concepts we developed to bring property-based testing to visualization development. The use cases we describe are just exemplary of the wide applicability of our approach.

The remainder of this paper is structured as follows. We first discuss related work in Section 2, before introducing relevant prerequisites regarding Inviwo and property-based testing in Section 3. We detail our approach to adapting property-based testing for vi-

ualization development, on the example of Inviwo, in Section 4. In Section 5 we show possible use cases for our approach before we conclude in Section 6. Our contributions can be summarized as follows:

- We propose a concept to integrate property-based tests into visualization development.
- We improve upon the current state of testing implemented in visualization software, which only covers the parameter space of the visualization algorithms sparsely, as we sample larger portions of this space efficiently through automatic test generation.
- Our approach is compatible with Inviwo’s existing regression test system, which enables execution of tests whenever a code change is pushed to the public repository.

2. Related Work

In this section, we give an overview of prior research related to our approach, mostly from the field of verifiable visualization.

Verification is the process of assessing the correctness of a numerical solution to a given model [Roy05]. In the context of visualization, this usually means assessing the correctness of renderings against the underlying mathematical models. For example, a volume visualization algorithm can be considered correct if its results match the volume rendering integral [Max95]. The visualization community has started recognizing the need for verification in visualization already in the 90s [GU95]. Kirby and Silva [KS08] also advocate for a common framework for verification in visualization and introduce the term *verifiable visualization*. The verification process is typically divided into *solution verification* and *code verification* [KS08].

Solution verification focuses on evaluating the accuracy of the resulting visualization with regard to the continuous solution given by the volume rendering equation, and thus investigates the *extent* of errors introduced through numerical approximation. Note that many of the techniques used to verify visualization solutions are also well known in computer graphics [UWP06].

Etiene et al. [EJR*13] investigate the correctness of a rendering with respect to the discretization of the volume rendering integral. They progressively refine step, grid and pixel sizes to analyze the convergence towards the continuous solution for direct volume rendering algorithms. In another work [ESN*09], they also inspect the convergence rates of iso-surface features like surface normals, area and curvature for several iso-surface approaches using the method of manufactured solutions.

Zheng et al. [ZXM10] provide a framework to measure and correct errors in the reconstruction of projection-based data, like CT. More specifically they measure and bound the errors introduced through interpolation of off-grid data samples.

Code verification on the other hand focuses on detecting mistakes in the source code or the algorithmic implementation. Prior work on code verification in visualization is rather sparse, despite its wide-spread use in other disciplines of computer science. We are not aware of any published papers on the subject, however we can inspect existing visualization software to get an overview of the current state of code verification in visualization.

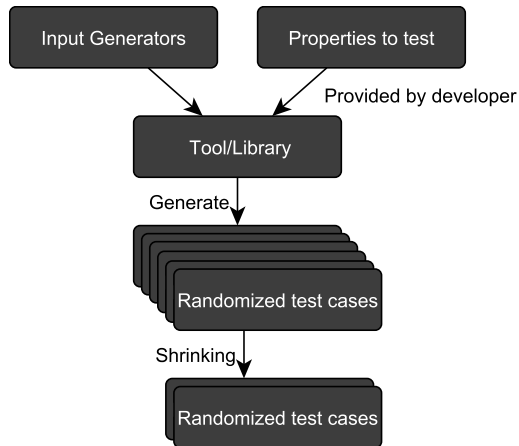


Figure 2: The typical components of property-based testing.

Looking at open source visualization software, we found that VTK [SLM04], VisTrails [CFS*06], VisIt [Chi12] and Inviwo [JSS*19] all use unit tests and regression tests. Unit tests are used to verify the behaviour of individual code classes or modules, while regression tests are used to find errors in the integration of such classes or modules. This is always done by producing a rendering for a fixed set of visualizations, that is then compared to a reference image produced when creating the test. However, those regression tests always run with a single parameter configuration and can miss bugs that only occur for a different set of parameters. This problem can be solved with property-based testing, which allows for an automatic check of the visualization with many different configurations, by using an efficient sampling strategy for the parameter space.

3. Prerequisites

In the following subsections, we will describe a few prerequisites regarding Inviwo and property-based testing.

3.1. Property-Based Testing

Property-based testing [FB97] is a testing technique that uses randomly generated test cases to test specified properties. This technique allows for tests to be written declaratively. This means that the author of a test case only needs to describe the expected properties and a tool takes care of generating actual test cases to test those properties.

The general flow of property-based testing is shown in Figure 2. Typically, a property-based testing tool allows the definition of input generators, but also provides default generators for common types such as integers, floats, or lists of integers/floats. The tool uses the input generators to generate random input data, and tests the *system under test*, e.g., an implementation of an algorithm against this data to see if the specified properties hold true. After a generated test case identifies an error, i.e., the result does not satisfy a specified property, the tool simplifies the inputs as long as the generated test cases still fail. Eventually, this results in a minimal failing example. This process is called *shrinking*.

For example, if we wanted to test a sorting algorithm, we could specify that after sorting a list with our function, the resulting list should contain the same elements (likely in a different order) as before sorting. For this example, the tool would generate random lists and check for each of them the property we specified, i.e., that the resulting list must contain the same elements as the input list. If our implementation fails in handling duplicate values, the result of sorting the list [4, 2, 3, 1, 3] could be [1, 2, 3, 4]. Now, that we have detected the error with this test case, the shrinking process removes elements from the list and checks to see if the error still occurs. After the whole shrinking process, the reported list would be [3, 3] which would erroneously get sorted to [3] by the faulty implementation. So by shrinking, we have obtained a minimal test case that already reveals the error in our implementation.

The best known implementation of property-based testing is QuickCheck [CH00] for the functional programming language Haskell. In QuickCheck, functions are used to define the expected properties. For existing types there are default generators that can generate random values. For user-defined types, as well as for existing types, QuickCheck allows for specifying custom generators. Following the success of QuickCheck for Haskell, its concepts have been adopted by libraries for several mainstream programming languages, such as C [GKS05], C++ [ee], and Java [PLS19].

In this paper, we describe how we adapt these concepts to enable property-based testing for visualization development. To avoid confusion in the terminology, we refer to the testable properties, in our case expected image changes, as *effects* and reserve the word *property* for Inviwo properties.

3.2. Inviwo

In this section we briefly introduce Inviwo [JSS*19] and its parts and concepts that are relevant to our implementation of property-based testing, as well as an overview of its existing testing pipeline. Inviwo is an extensible interactive visualization system that provides several usage abstraction levels. These abstraction levels range from developing low-level device-specific C++ code to modeling data-flow interactively at a high-level in Inviwo's GUI. While the lower levels are used to extend the system with new capabilities through *modules*, the GUI can be used to create visualizations by using a network editor. The network resembles a directed acyclic graph with *processors* as nodes, representing functional units, and data-flow as edges, transferring data from a processor's *outport* to another processor's *inport*. A screenshot of the GUI showing such a network can be seen in Figure 3. The parameters of an algorithm which is implemented by a processor can be exposed using *properties* (see the window on the right in Figure 3). Processors can implement functions such as for instance data loading, filtering, rendering and displaying, allowing users to build custom visualization pipelines interactively using drag-and-drop. Properties of a processor can be all types of parameters associated to the processor's function, ranging from file paths, over lighting parameters to transfer functions.

Testing in Inviwo is currently implemented through unit tests to validate individual modules and processors, as well as integration tests to validate a correct integration of multiple different proces-

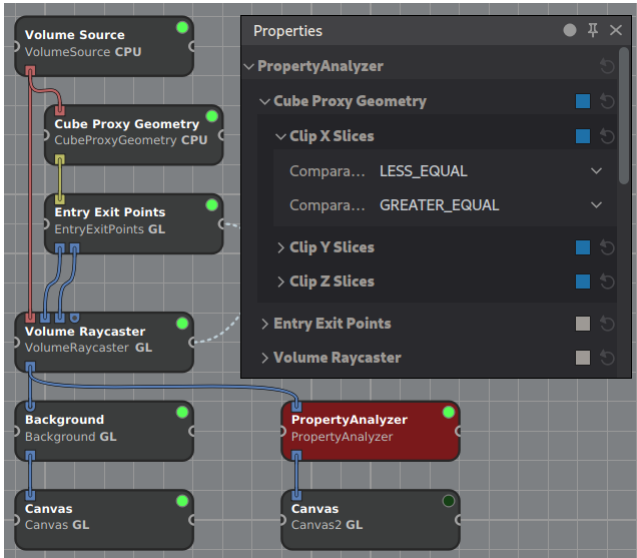


Figure 3: The GUI to create tests with our implementation. We can define effects for all compatible properties from the previous processors of the network. In the case of Clip X Slices we expect the resulting number of pixels with background depth to be less or equal, when increasing the lower bound, and greater or equal when reducing the upper bound. The tested renderings of the clipped volume can be seen in Figure 6.

sors in common scenarios. Inviwo allows for automatically generating regression tests from a given network, using the currently displayed canvas images as references to compare against in test runs, which are executed on a continuous integration server. Thus, code changes that break existing functionality can be easily spotted before being merged into the public repository. On top of that, Inviwo allows for visual debugging inside the GUI by providing means to inspect common data types at any place in the network by hovering over an input or an output. This allows for easy checking of intermediate results while developing a visualization.

One major drawback of Inviwo’s current testing capabilities is that the aforementioned regression tests can only find errors occurring with the property values specified when creating the test. Therefore, the current approach will miss errors that only occur with different network parameters, i.e., with different values for the processors’ properties. This issue is addressed with our presented approach, which employs property-based testing to also cover this parameter space with tests.

4. Property-Based Testing in Visualization Development

In this section, we detail our approach to enable property-based testing in visualization development. We first describe the underlying ideas and concepts, and after that we explain how we implemented them.

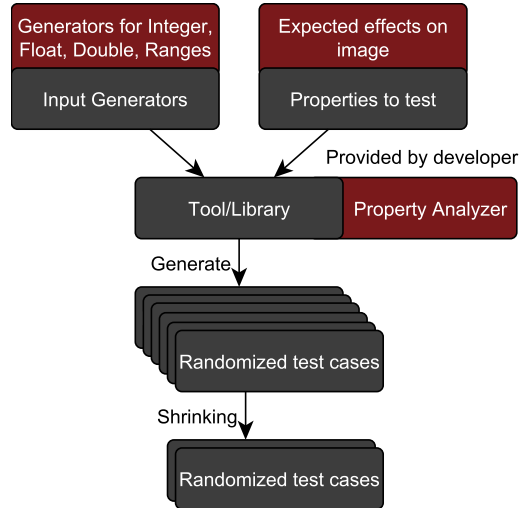


Figure 4: Illustration of our adaptation of the components of property-based testing when applied to visualization development. Our contributions are emphasized in red.

4.1. Ideas & Concepts

Since test-driven development is cumbersome to realize with current visualization software, due to the requirement for expected images, we decided to adapt the concepts of property-based testing to visualization development. Figure 4 illustrates how we adapt each part of property-based testing.

It is possible to test processors individually, but normally processors are not used alone. They are part of a network that consists of several processors. So to test a processor, its interaction with other processors must also be tested. Therefore, to generate test cases for property-based testing, we randomize the properties of the processors in the network (analogous to the **input generators** of property-based testing). While randomizing the properties alone does not make test cases, we also need a way to specify the effects that we expect on the resulting images caused by differing values of properties (analogous to the **properties to test** of property-based testing). Our implementation, the **Property Analyzer**, is conceptually described in the following, with additional implementation-specific details described in Section 4.2.

In order to test an effect on a rendered image, we first investigate aggregated image metrics. For that, we calculate a *score* for each generated image, that is, we map the image to some (real or integral) number. This allows us to perform comparisons such as *equal*, *less than*, *greater than*, and so on to measure the effects of a changed property. This can easily be extended with additional scores or even with per pixel comparisons. These comparisons should be interpreted as the expected effect, obtained when increasing the value of a property of the processor. For example, the *less than* comparison means that an increase in the value of the property should result in a decrease of the score. If this constraint is not true, i.e., the score does not decrease, we have identified an *error* because the specified effect is not satisfied. For our example

of clipping planes, we determine the number of pixels with a given depth. The idea is that if there is a background and an object, all the pixels of the background have the same depth, and thus the number of pixels with that depth are the pixels that are not obscured by the object that is in the foreground. Changing the clipping planes will cause more fragments of the object to be discarded, and thus more pixels of the background to be visible.

In contrast to the testing methods currently existing in Inviwo, such as its unit tests, with property-based testing a user does not explicitly specify the properties of the processors for a test case, but specifies the expected effects for different values of a property. We let the user specify which properties should be tested and which effect in form of a comparison operator is expected by increasing the property. While property-based testing allows to define additional (randomized) unit tests, it also allows integration tests. In this context, property-based integration tests have the advantage that they can be defined from visual expectations alongside a visualization, using the same network, without the need to write code. As visualization development usually produces an image, it is natural to test visualization algorithms based on their effects on the resulting image.

In order for a property to be testable, there must be implementations for generating and comparing values or components of values. For the purpose of this paper, we provide default implementations for properties with a single integer or a single floating point value or a range of such. Upon testing, we generate up to nine values for each chosen testable property and combine them into sets of assignments. These nine values include numerical limits of the data type as well as uniformly chosen random samples within those limits.

In a network there can be several processors, each of which has several properties. Since several random values are generated for each of these properties, a huge number of combinations is possible. By restricting the combinations to pairs, this number is significantly reduced. For each chosen property we generate *tests*, such that each possible pair of values for different properties is in some test. That means the generated set of tests satisfies *pairwise coverage*. Two tests are *comparable*, if and only if the expected effects of the individual properties are non-contradictory, that is, there is a *combined expected effect* that implies each individual effect. For example, in the case of our clipping example, if there is a property that controls the opacity of the rendered object. A value of 0 would make the object completely invisible and thus make the clipping have no effect anymore.

For the generation of pairwise coverage, we use an algorithm for covering array construction, which in our case constructs an array containing test cases, so that each pair of random values is covered. At first, we constructed our covering array using the discrete SLJ strategy from [SC19]. Typically, the goal of algorithms for constructing covering arrays is to keep them as small as possible, and this algorithm achieves that quite well, but it turned out that these minimal sets unfortunately also contained only very few comparable pairs of tests.

Since for our purposes, the number of *comparable pairs* of tests should be as large as possible (for a fixed number of tests), we developed our own greedy algorithm which not only aims to keep the

total number of tests low, but simultaneously also aims to maximize the number of comparable pairs of tests. Let A_1, A_2, \dots, A_t be the sets of possible assignments for the tested properties, and let $A'_i = A_i \cup \{\perp\}$, where \perp represents an undefined assignment. We call the elements of $\mathcal{A} = A_1 \times \dots \times A_t$ (*complete*) tests and the elements of $\mathcal{A}' = A'_1 \times \dots \times A'_t$ *partial tests*, and we say a (partial) test $a = (a_1, \dots, a_t)$ *implies* another (partial) test $b = (b_1, \dots, b_t)$, if and only if $b_i \in \{\perp, a_i\}$ for all $i \in \{1, \dots, t\}$, and a and b are *disjoint*, if and only if $(a_i \neq \perp) \wedge (b_i \neq \perp) \implies a_i = b_i$ for all i . We say $c = (c_1, \dots, c_t)$ arises from *extending* a with b , where $c_i = a_i$ if $a_i \neq \perp$ and $c_i = b_i$ otherwise. Note that if a and b are disjoint, c implies both a and b . Our general approach is shown in Algorithm 1. When successively constructing our covering array $C \subseteq \mathcal{A}$, we first generate all possible interactions I ,

$$I = \{(a_1, a_2, \dots, a_t) \in \mathcal{A}' : |\{a_i \neq \perp : i \in \{1, \dots, t\}\}| = 2\},$$

between two properties, and while there are still *unused* (pairwise) interactions (i.e. interactions in I that are not implied by some already generated test), we construct a new test $v \in \mathcal{A}'$ which initially is equal to one randomly selected unused interaction. While v is not complete, we extend it with either a disjoint and not implied unused interaction or a (not necessarily disjoint) previously created test, such that the resulting partial test maximizes the sum of weights of all comparable previously created tests. Here, the *weight* of a test is equal to the current number of created tests to which it is not comparable. This is intended to increase the minimum number of other generated tests, that a test is comparable to, such that there are as few tests as possible for which there are no other comparable tests. Also, the weight of unused interactions is multiplied by a given constant in order to "encourage" using multiple uncovered interactions in one test and thereby reduce the total number of tests. Note that for any incomplete test there is always a (possibly already used) disjoint interaction, so our algorithm is guaranteed to terminate and output a 2-covering array.

Since there can be many processors in a network, each of which can have multiple properties, there can be an unmanageable amount of randomizable values. It would be difficult to determine which combination of parameters actually caused an identified error. To solve this, we adapted the concept of shrinking, as introduced in Section 3.1. When an error is identified, we reset the randomized properties one by one to their original value and test to see if an error still occurs. In the end, we will have a minimal set of randomized properties that still causes an error.

The concepts described above can easily be adapted to any other rendering pipeline that exposes its properties in a way that allows them to be varied programmatically. In the following we describe our exemplary implementation of these concepts as an Inviwo processor.

4.2. Implementation

We have implemented property-based testing in Inviwo by realizing a processor called *PropertyAnalyzer*. This processor receives the image produced by the network that we want to verify, and gathers the modifiable properties from all processors involved in producing that image. These gathered properties can then be used to define tests through the properties of the *PropertyAnalyzer* processor as

```

input : Sets of possible assignments  $A_1, \dots, A_t$ ,
        weight multiplier  $m$ 
output: A 2-covering array
 $C \leftarrow \emptyset$ ;
while there is uncovered interaction  $u$  do
   $v \leftarrow u$ ;
  while  $v \notin \mathcal{A}$  do
     $w_{opt} \leftarrow -\infty$ ;
     $tmp \leftarrow \perp$ ;
    for uncovered interaction  $u'$  do
      if  $disjoint(v, u') \wedge \neg implies(v, u') \wedge m \cdot$ 
         $wS(C, extend(v, u')) > w_{opt}$  then
        |  $tmp \leftarrow extend(v, u')$ ;
        |  $w_{opt} \leftarrow m \cdot wS(C, tmp)$ ;
      end
    end
    for  $c \in C$  do
      if  $wS(C, extend(v, c)) > w_{opt}$  then
      |  $tmp \leftarrow extend(v, c)$ ;
      |  $w_{opt} \leftarrow wS(C, tmp)$ ;
    end
     $v \leftarrow tmp$ ;
  end
   $C \leftarrow C \cup \{v\}$ ;
end
return  $C$ ;

```

Algorithm 1: Our algorithm for the generation of the covering array. With $weight(C, a) = |\{c \in C : \neg comparable(c, a)\}|$ being the number of tests in C that are not comparable to a and $wS(C, a)$ being the sum of the weights of all tests in C that are comparable to a , i.e. $wS(C, a) = \sum \{weight(c) : c \in C, comparable(c, a)\}$

can be seen in Figure 3, which shows the GUI to create tests for our clipping example. When executing the tests, the PropertyAnalyzer processor changes the selected properties and verifies whether the resulting changes in its input image comply with the desired effects set by the user.

The PropertyAnalyzer further has an image output. This output shows a completely white image if and only if all the tests pass. As Inviwo’s current regression testing system relies on image comparison to determine the success of a regression test, we can easily use this image to automatically generate a regression test that executes our property-based tests. This allows us to include the property-based tests into Inviwo’s continuous integration pipeline, that executes all tests whenever a code change is pushed to the public repository.

Currently, we calculate a *score* for each image arriving at the PropertyAnalyzer. For now, the score is either the number of pixels with a given color or the number of background pixels (i.e., those with depth value of 1). Note, that while our implementation currently only supports this aggregated score, the system can easily be extended to pixel-wise comparisons as well.

The test creation GUI has a checkbox and a drop-down menu for

each testable property, which allows the user to specify whether the property should be included in the testing and what the expected effect of increasing its value has on the rendering. Possible choices for the expected effect that an increase of a property’s value may have on the score include the common numerical comparisons ($=, \neq, <, \leq, >$ and \geq), as well as ANY and NOT_COMPARABLE. The latter two signal that the respective property should have no effect on the score, and that scores from tests where this property has different values are not meaningfully comparable, but the network should be tested with different values for this property nevertheless. The NOT_COMPARABLE comparison can be used, for example, to vary both the input data and the camera position. In the clipping example, we specify that increasing the X position of the lower bound clipping plane (upper drop-down in Figure 3) should result in fewer or equal pixels with depth value 1. Similarly we expect this score to become greater or equal when we increase the upper bound position (lower drop-down in Figure 3). Our implementation conveniently generates a textual description of the tests as well, see Figure 5.

To limit the time of the test run, we also allow for setting a maximum number of tests that may be executed. Note that, if this number is lower than the number of tests that would be created by our algorithm, the resulting test set does *not* satisfy pairwise coverage.

As described in Section 4.1, for the generation of pairwise coverage, we first implemented the discrete SLJ strategy from [SC19] to construct the covering array. This algorithm succeeded in constructing a small covering array but since for our purposes, the number of comparable pairs of tests should be as large as possible (for a fixed number of tests), we developed and implemented a greedy algorithm which aims to keep the total number of tests low, while simultaneously maximizing the number of comparable pairs of tests. Table 1 shows an exemplary performance comparison between the discrete SLJ strategy and our algorithm. In particular, note that the number of comparable test pairs steadily decreases for the SLJ algorithm when we increase the number of tested properties, while it generally increases for ours, even when we ensure that the number of tests is equal for both algorithms.

When there is a pair of tests for which the combined expected effect contradicts the observed effect, we have found an error, that is, a case where the network does not behave as specified by the user. In order for the user to evaluate the errors, we generate a report document (in HTML) containing all property values as well as the

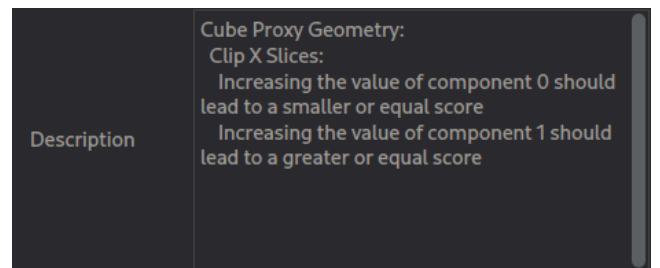


Figure 5: Our PropertyAnalyzer processor generates a textual description of the clipping test.

#P	#I	SLJ		OURS		
		#T	#C	#T	#C	limited
2	81	81	1177	81	1177	1177
3	243	129	1356	195	3951	2021
4	486	170	974	217	4440	2868
5	810	199	824	348	11827	5362
6	1215	217	407	328	6485	4466
7	1701	232	106	440	13308	6750
8	2268	259	115	485	12361	6945
9	2916	269	49	579	18453	8388

Table 1: Exemplary results of the discrete SLJ strategy and our covering array algorithm on numerous IntMinMax-properties, whose comparators were all set to `GREATER_EQUAL` and `LESS_EQUAL` for their lower and upper bounds, respectively. The weight multiplier for `ours` was set to 5.0. Here, #P and #I are the number of tested properties and generated interactions, the #T and #C columns contain the number of generated tests and number of comparable test pairs, and the `limited` column for `ours` contains the number of comparable test pairs, when the number of tests is limited to the number of tests generated by SLJ in the same row.

resulting image for both tests for each error. Figure 6 shows an excerpt of such a report for a faulty implementation of a clipping processor. With the generated report, it is easy to compare test runs to identify errors, however, this report shows all tests that failed. Especially, in a test-driven scenario this may result in very long, hard to overview reports.

To deal with this problem, we have also implemented functionality to *distill*, i.e., shrink, the set of tested properties down to a minimal subset that leads to an error, so that the faulty network part can be located more easily. This is simply done by successively removing properties from the set of tested properties whenever there would still be errors when testing the remaining properties. In the end, the minimal test case reveals at least one error, but not necessarily all errors that were revealed in the original test case. However, this is not a problem because after fixing the errors that got revealed by the minimal test case, the tests should be rerun anyways and will eventually reveal the other errors in one of the minimal test cases generated by the following test runs. For this reason, the order in which the properties are removed from the set of tested properties only affects the order in which errors are revealed but does not affect which errors will be revealed in the end.

In summary, our implementation allows to automatically gather all comparable properties that we might want to test. We can easily define what effect we expect from changing a property’s value, from within the GUI. Our approach automatically generates a set of tests that is efficient with regard to error detection rate and presents the results in a report. By distilling the test results, we can pin-point errors quickly without having to look through all the failed tests.

5. Use Cases

In this section, we illustrate the effectiveness of property-based tests for developing visualization techniques. Therefore, we de-

Property Values				Property Values			
DisplayName	Value			DisplayName	Value		
Cube Proxy Geometry	Cube Proxy Geometry	199651	LESS_EQUAL	163606	Cube Proxy Geometry		Equal Values
	Clip X Slices [0, 78]				Clip X Slices [0, 149]		

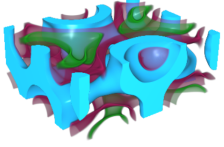
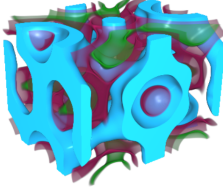



Figure 6: Example for a test report, showing the two property configurations and their associated rendering that made the test fail. In the middle of the top row, the amount of background pixels is shown for each configuration and how they should compare according to the test specification.

scribe several use cases where property-based testing can help finding implementation errors. The clipping use case described throughout this paper, is presented in Section 4. Note, that after integrating the presented property-based test use cases into Inviwo’s regression test system, they can be verified on the continuous integration server after every code change pushed to the public repository.

5.1. Lighting Parameters

Another excellent use case for property-based testing is varying lighting parameters to find errors in the rendering algorithm. Although not all parameters of a rendering technique have an easily testable effect on the resulting image, we can capture some common effects that should apply to all rendering techniques. In general, increasing or decreasing light intensity results in a monotonic increase or decrease in image brightness respectively.

A specific example with the Phong shading parameters - ambient, diffuse and specular lighting - is illustrated in Figure 7. A correctly implemented rendering algorithm using this model should always produce an image with the same or higher/lower pixel-wise brightness when increasing/decreasing these parameters. Note that this effect is true for differences in one of those components at a time, but not necessarily when comparing two renders where multiple of the components vary in different directions, i.e., one with low ambient and high diffuse and another vice versa.

In order to test for that with our implementation, one would first need to implement pixel-wise comparison. Given that, a test can be designed using an individual *PropertyAnalyzer* processor for each of the lighting parameters, ambient, diffuse and specular, respectively. Each *PropertyAnalyzer* verifies the effect for its lighting property, while setting the others to `NOT_COMPARABLE`, in order to avoid tests with parameter sets that vary in multiple lighting properties simultaneously. Setting the other components to `NOT_COMPARABLE` will still generate tests with different values for those, but avoids the aforementioned issue by excluding comparisons where multiple components are changed at the same time.

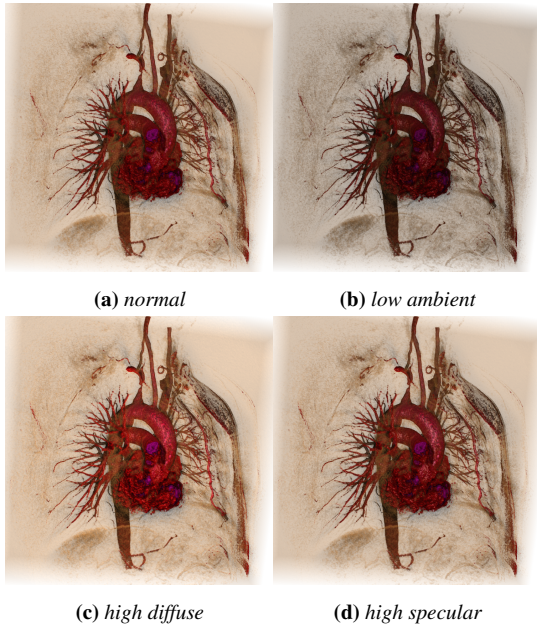
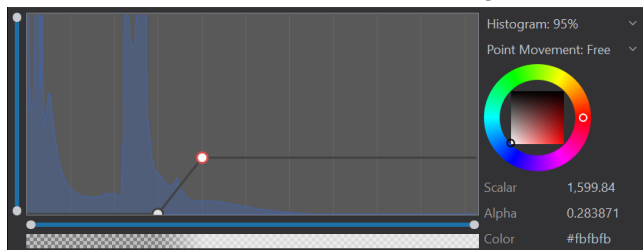
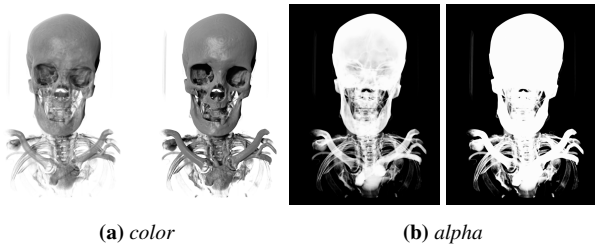


Figure 7: Increasing or decreasing the brightness of lights and materials should increase or decrease the brightness in the resulting image, respectively.



(c) Transfer function used for the transparent rendering (left). The opacity of the selected control point is increased to get the opaque rendering (right).

Figure 8: Increasing or decreasing the opacity of the transfer function should result in an increase or decrease in alpha, respectively.

5.2. Compositing

Manually finding errors in the integration of light along a ray can be very difficult, especially when transparency effects are involved. While it is impossible to test all aspects of this complex task, there are again common assumptions that we can make, that should apply to all of our integration approaches. One of those assumptions is that, when increasing the opacity of the material, as defined by a transfer function, the resulting image should monotonically increase in opacity. This effect is illustrated in Figure 8.

In order to test that, we can manipulate the transfer function. In Inviwo transfer functions can be defined as piece-wise linear functions using control points in a widget, as illustrated in Figure 8c. After implementing an appropriate input generator for the transfer function, the PropertyAnalyzer can generate assignments with two transfer functions that vary in the opacity of a single control point. Each pair of renderings is then checked for our assumption, that a larger opacity in the transfer function should lead to an equal or higher pixel-wise alpha. Note, that this assumption only holds, when varying a single control point at a time, that means all other control points should be set to NOT_COMPARABLE.

5.3. Limitations

However we also found limitations of our approach. When relying only on the produced rendering to determine the correctness of an effect, we can miss implementation errors complying with the desired effects. In the clipping example, our approach would be unable to detect an error if two axes are switched, because clipping a different axis still has the same effect on the aggregated image metrics. Our approach should be used complementary to existing unit and regression testing strategies, in order to increase their parameter coverage, but it should not be seen as a replacement for other types of tests.

6. Conclusion

In this work, we propose the integration of property-based testing into visualization development. We argue that property-based testing can be used to vastly increase the coverage of different rendering parameters in the testing procedure. This allows us to find implementation errors quicker and increases the confidence in the correctness of our implementations. In fact, our proposed property-based testing implementation can even be used to create visualizations in a test-driven manner, by defining the tests alongside the actual algorithm. With this new testing strategy, we offer an easy way for visualization researchers and developers to create meaningful tests automatically from within a GUI, just by defining their expectations for parameter changes. By making the testing process easier and more expressive, we expect to further close the gap between research and software development in the field of visualization.

In the future, we plan to further extend the capabilities of our implementation through new types of effects, as well as more convenient ways to generate multiple, very similar tests on the fly, like the Phong lighting example from Section 5.1. Another interesting direction to follow is the development of an interface, that allows users to interactively choose additional reference images for regression tests, based on property-based testing parameter generation, as an easy way to cover a larger parameter space through regression tests (using image comparison) directly.

Acknowledgments

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under grant 391107954 (Inviwo). The proposed concepts have been realized using the Inviwo visualization framework (<https://inviwo.org>).

References

- [AAA*19] AKIYAMA K., ALBERDI A., ALEF W., ASADA K., AZULAY R., BACZKO A.-K., BALL D., BALOKOVIĆ M., BARRETT J., BINTLEY D., ET AL.: First m87 event horizon telescope results. iv. imaging the central supermassive black hole. *The Astrophysical Journal Letters* 875, 1 (2019), L4. doi:10.3847/2041-8213/ab0ec7. 1
- [Bec03] BECK K.: *Test-driven development: by example*. Addison-Wesley Professional, 2003. 1
- [CFS*06] CALLAHAN S. P., FREIRE J., SANTOS E., SCHEIDEGGER C. E., SILVA C. T., VO H. T.: Vistrails: Visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), SIGMOD '06, Association for Computing Machinery, p. 745–747. URL: <https://doi.org/10.1145/1142473.1142574>, doi:10.1145/1142473.1142574. 3
- [CH00] CLAESSEN K., HUGHES J.: Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, Association for Computing Machinery, p. 268–279. URL: <https://doi.org/10.1145/351240.351266>, doi:10.1145/351240.351266. 3
- [Chi12] CHILDS H.: Visit: An end-user tool for visualizing and analyzing very large data. 3
- [ee] EMIL E: rapidcheck - QuickCheck clone for C++ with the goal of being simple to use with as little boilerplate as possible. <https://github.com/emil-e/rapidcheck>. Accessed: 2021-03-09. 3
- [EJR*13] ETIENE T., JÖNSSON D., ROPINSKI T., SCHEIDEGGER C., COMBA J. L., NONATO L. G., KIRBY R. M., YNNERMAN A., SILVA C. T.: Verifying volume rendering using discretization error analysis. *IEEE transactions on visualization and computer graphics* 20, 1 (2013), 140–154. doi:10.1109/TVCG.2013.90. 2
- [ESN*09] ETIENE T., SCHEIDEGGER C., NONATO L. G., KIRBY R. M., SILVA C.: Verifiable visualization for isosurface extraction. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (2009), 1227–1234. doi:10.1109/TVCG.2009.194. 2
- [FB97] FINK G., BISHOP M.: Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (July 1997), 74–80. URL: <https://doi.org/10.1145/263244.263267>, doi:10.1145/263244.263267. 2, 3
- [GKS05] GODEFROID P., KLARLUND N., SEN K.: Dart: Directed automated random testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. URL: <https://doi.org/10.1145/1064978.1065036>, doi:10.1145/1064978.1065036. 3
- [GU95] GLOBUS A., USELTON S.: Evaluation of visualization software. *ACM SIGGRAPH Computer Graphics* 29, 2 (1995), 41–44. doi:10.1145/204362.204372. 2
- [JSS*19] JÖNSSON D., STENETEG P., SUNDÉN E., ENGLUND R., KOTTRAVEL S., FALK M., YNNERMAN A., HOTZ I., ROPINSKI T.: In-vivo - a visualization system with usage abstraction levels. *IEEE Transactions on Visualization and Computer Graphics* 26, 11 (2019), 3241–3254. doi:10.1109/TVCG.2019.2920639. 2, 3
- [KKF*17] KOZLÍKOVÁ B., KRONE M., FALK M., LINDOW N., BAADEN M., BAUM D., VIOLA I., PARULEK J., HEGE H.-C.: Visualization of biomolecular structures: State of the art revisited. In *Computer Graphics Forum* (2017), vol. 36, Wiley Online Library, pp. 178–204. doi:10.1111/cgf.13072. 1
- [KMM*18] KREISER J., MEUSCHKE M., MISTELBAUER G., PREIM B., ROPINSKI T.: A survey of flattening-based medical visualization techniques. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 597–624. doi:10.1111/cgf.13445. 1
- [KOJC13] KERSTEN-OERTEL M., JANNIN P., COLLINS D. L.: The state of the art of visualization in mixed reality image guided surgery. *Computerized Medical Imaging and Graphics* 37, 2 (2013), 98–112. doi:10.1016/j.compmedimag.2013.01.009. 1
- [KS08] KIRBY R. M., SILVA C. T.: The need for verifiable visualization. *IEEE Computer Graphics and Applications* 28, 5 (2008), 78–83. doi:10.1109/MCG.2008.103. 1, 2
- [KWG04] KUHN D. R., WALLACE D. R., GALLO A. M.: Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421. doi:10.1109/TSE.2004.24. 1
- [Max95] MAX N.: Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics* 1, 2 (1995), 99–108. doi:10.1109/2945.468400. 2
- [PLS19] PADHYE R., LEMIEUX C., SEN K.: Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2019), ISSTA 2019, Association for Computing Machinery, p. 398–401. URL: <https://doi.org/10.1145/3293882.3339002>, doi:10.1145/3293882.3339002. 3
- [PSK*10] PERROUIN G., SEN S., KLEIN J., BAUDRY B., L. TRAON Y.: Automated and scalable t-wise test case generation strategies for software product lines. In *2010 Third International Conference on Software Testing, Verification and Validation* (2010), pp. 459–468. doi:10.1109/ICST.2010.43. 1
- [RE07] RUEDEN C. T., ELICEIRI K. W.: Visualization approaches for multidimensional biological image data. *Biotechniques* 43 (2007), S31–S36. doi:10.2144/000112511. 1
- [Roy05] ROY C. J.: Review of code and solution verification procedures for computational simulation. *Journal of Computational Physics* 205, 1 (2005), 131–156. doi:10.1016/j.jcp.2004.10.036. 2
- [SC19] SARKAR K., COLBOURN C. J.: Two-stage algorithms for covering array construction. *Journal of Combinatorial Designs* 27, 8 (2019), 475–505. doi:10.1002/jcd.21657. 5, 6
- [SLM04] SCHROEDER W. J., LORENSEN B., MARTIN K.: *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004. 3
- [UWP06] ULBRICHT C., WILKIE A., PURGATHOFER W.: Verification of physically based rendering algorithms. In *Computer Graphics Forum* (2006), vol. 25, Wiley Online Library, pp. 237–255. doi:10.1111/j.1467-8659.2006.00938.x. 2
- [WK02] WALLACE D., KUHN D.: Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering* 08 (07 2002). doi:10.1142/S021853930100058X. 1
- [ZXM10] ZHENG Z., XU W., MUELLER K.: Vdvr: Verifiable volume visualization of projection-based data. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (2010), 1515–1524. doi:10.1109/TVCG.2010.211. 2