Visualising Linear Algebra Operations (revised April 2023)

Student Name: T.J.R. Stokes
Supervisor Name: T.K. Friedetzky
Submitted as part of the degree of BSc Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—Linear Algebra is an essential field in many areas of computer science, such as game development and search engine development, but its operations can seem unintuitive to students new to the field. This is why the project aimed to provide a tool for visualising common linear algebra operations, such as matrix multiplication and the calculation of eigenvectors.
The user could define the transformation of a shape using a matrix, which was applied using matrix multiplication, and generate the eigenvectors of this matrix. Each element of it could be a mathematical expression which was parsed into an expression tree using the LR(1) parsing algorithm. This tree could evaluate itself as a float for the purpose of passing data to the graphics API. The user could also choose between a two dimensional or three dimensional view, and between a linear or affine transformation. These two choices controlled the size of the transformation matrix being used.
For the two dimensional view the grid and the fixed view point made the transformations and eigenvectors very clear for each matrix, and the optional animation of the transformation helped the user to understand their meaning. However, for the three dimensional view only certain matrices were clear as for points lying far from the x-z grid there was no frame of reference for their location. This limited the extent to which the user could improve their understanding.

Index Terms—Data and knowledge visualization, Eigenvalues and eigenvectors, Numerical Linear Algebra, Parsing

1 Introduction

Linear algebra is a vital field of mathematics with uses that span a vast amount of computer science. Its main focuses are the study of linear equations, and linear maps. An important example of the application of linear algebra in computer science is in algorithms used to rank web page results by search engines. Google's PageRank algorithm forms a matrix where each element ij is the ratio of links from i to j, to the total number of links from i. It is clear from this that the dimension of the matrix formed would be equivalent to the number of pages being considered, so in the case of the world wide web the dimension of this matrix is in the millions. An example of the matrix used when the PageRank algorithm was performed on the Cambridge university network is seen in figure 1 to demonstrate the scale that these algorithms can reach.

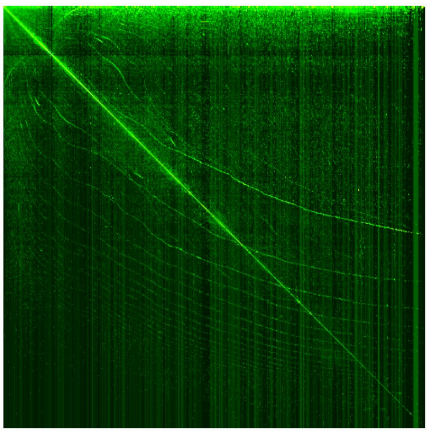


Fig. 1. This shows the matrix constructed when the PageRank algorithm was used on the Cambridge University network. Its dimension is 212710. [1]

Tools called Computer Algebra Systems allow for expressions to be manipulated in a similar way to manual computations performed by mathematicians. A popular example of this is Wolfram Alpha. There are also popular programming libraries designed to work only with basic data types, such as floats, as inputs for the matrix components such as OpenGL Mathematics (GLM) [2]. GLM is optimised for small matrices for the purpose of graphics development. Similar tools exist for large matrices such as Eigen [3], which takes advantage of specific properties of the matrix, such as whether it sparse or not, to use the most efficient algorithm. The key feature that these tools lack is some visualisation of the operations being performed.

Understanding linear operations from examples such as the PageRank algorithm is challenging, due to the complexity of the computation and the lack of visual representation of higher dimensions. This is why examining the role of smaller matrices in computer graphics is useful for improving understanding. In this field matrices are used to transform vectors representing vertices. In this way each matrix can be viewed as a transformation such as scaling, translating or skewing. or as a product of these transformations. This kind of representation could hep students new to the field in understanding matrix multiplication. This benefits of this representation extend to other operations such as the calculation of eigenvectors. Since any vertex lying on an eigenvector would only be scaled radially, an animation

of a matrix transformations would make them very clear, especially if the user can choose to highlight them. This project aimed to provide a program that demonstrated a clear visualisation of this representation.

1.1 Motivation

In the field of computer graphics linear algebra holds an important role. This is because each vertex has to be transformed according to its position in the world, the position of the camera, and the type of projection used, in 3 dimensions. The most effective method of performing these transformations is to represent each vertex location as a vector then to perform matrix multiplications. One caveat of this is that translations must be possible, which are not linear transformations since the position of the origin is not preserved, but the underlying theory remains the same.

A lack of understanding of why certain matrices represent certain transformations can be of particular difficulty due to lack of standardisation in certain areas. For example, the look at matrix changes based on the coordinate system used (LH or RH), so someone without knowledge of the matrix's construction may struggle even if provided with a working maths library.

A big problem when studying Linear Algebra is that many aspects of it are hard to grasp when only the theory can be seen. In these cases some form of visualisation of the operation could be key in helping to develop understanding of the topic. This is inspired by the animations of 3Blue1Brown [4], and by the graphing calculator Desmos [5]. Desmos demonstrates the intuitive learning about graph transformations that is made possible by having access to a responsive visualiser, and 3Blue1Brown shows how there are many areas in Linear Algebra that would benefit from a similar publicly available tool.

1.2 Objectives

The basic goals of the project were to create a visualiser for matrix transformations that can operate in both 2 and 3 dimensions, with a suitable camera controller for each. In the n dimensional view, the transformation matrix can either be n by n, for a linear transformation, or n+1 by n+1 for an affine transformation. Various shapes should also be able to be modified, such as in 3 dimensions a cube, a sphere and a grid.

Further goals were to display the eigenvectors of the transformation matrix, and to animate the transformation so that the meaning of the eigenvector can be intuitively understood. To aid in the user experience common transformation matrices can be generated, such as rotation with the angle being a variable that can be controlled by the user. Similarly the user should also be able to generate the inverse of the current transformation matrix. A second transformation matrix, the model transform, should also be optionally modified. This transformation would be applied first, allowing the shape being transformed to be further controlled by the user. For example, this could be used to transform a cube into a frustum in order to observe the effect of a perspective projection.

The most challenging goals of the project were to allow the user to input expressions instead of just numbers. These expressions would be parsed into an expression tree, with the user being informed of invalid inputs. These expressions should be able to contain variables, functions, and basic mathematical operators such as addition and multiplication. The expression tree should also be simplified whenever possible, with the simplified form being converted back to a string and displayed to the user.

The final outcome of this project was intended to be a proof of concept of the idea that visualisation improves understanding. Because of this the user interface was rudimentary, with its only goal being to provide a mechanism for user input, not to create a visually appealing program. However, it must still provide some guidance to the user on how the program should be used.

2 Related Work

The aim of this project was to create computer algebra system, which is a piece of software that solves mathematical problems. They may choose to focus on specific areas, such as Desmos which has a focus on graphing of functions, or they may be more general such as Mathematica [6].

The remainder of this section examines the algorithms available for the mathematical operations required, the methods used to render the visualisation, then finally the parser used to generate expression trees.

2.1 Linear Algebra Computation

The main focus of this project is on the transformation represented by a matrix. This transformation is applied through matrix multiplication with a vector, so the algorithm chosen for this is important.

Strassen's algorithm is the fastest algorithm for matrix multiplication when matrices are large, as it is O($n^{2.81}$) [7], compared to O(n^3) for the brute force method. Since its creation there have been asymptotically faster algorithms, such as the Coppersmith-Winograd algorithm, which are not used in practice since they are only faster for matrices that are too large for current processors [8]. For 4x4 matrices however, which are the primary focus in this project, Strassen's algorithm offers no increase in speed [9], and due to the creation of extra matrices it also uses more memory, so there is no need for its increased complexity.

[9] also suggests that Single Instruction Multiple Data (SIMD) processing shows little speed improvement for smaller matrices, but this may be affected by an inefficient memory layout caused by attempting to use similar code for larger matrices. It's use is limited even more since expressions trees are used as elements of matrices and vectors as opposed to floats in most instances, but it will still be used in matrices and vectors with a floating point underlying type.

Observing these transformations is helpful for understanding eigenvectors, as points that lie on these vectors will only be transformed radially, but it would still be unclear without the eigenvector being clearly highlighted. For this reason the eigenvector is calculated for each matrix the user inputs. For 2-by-2 and 3-by-3 matrices the equation $(A - \lambda I)\mathbf{v} = \mathbf{0}$ is solved using Gaussian Elimination.

The numerical stability of Gaussian Elimination is a potential cause of concern for the accuracy of the eigenvector

computation, as there are some situations where the solution of an equation $A\mathbf{x} = \mathbf{b}$ for a matrix A where the computed solution is drastically different to the true solution as a result of rounding. An example of this is the LU factorisation of $A = \begin{bmatrix} 1 & 0 \\ 0.0001 & 0 \end{bmatrix}$ given in [10], which aims to reduce the matrix A into a lower diagonal matrix multiplied by an upper diagonal matrix.

With partial pivoting the growth factor of an n dimensional matrix is at most 2^{n-1}, which is still larger than desired, but in practice the growth factor has been observed to be much lower for the majority of matrices. Complete pivoting improves on this further, and it was conjectured for a time that the growth factor was at most n for an n dimensional matrix, but this was disproved by counterexamples of 13-by-13, 14-by-14, 15-by-15 and 16-by-16 matrices that exceeded this bound [11].

For larger matrices this approach's effectiveness is limited because for an n-by-n matrix, an n degree characteristic polynomial must be solved to calculate λ. In these cases, the QR algorithm[10] can be used to convert the matrix into a diagonal matrix, whose eigenvalues are the entries on the diagonal. This algorithm is more numerically stable than solving a higher order polynomial.

Given a matrix A, we set $A_0 = A$. For each step k we compute $A_k = Q_k R_k$ where Q_k is an orthogonal matrix and R_k is an upper triangular matrix. We then set $A_{k+1} = R_k Q_k$. It is important to note that A_{k+1} is similar to A_k so it has the same eigenvalues. This process is repeated for n steps until A_n is an upper triangular matrix.

2.2 Visualisation

The main contribution of this project is in its visualisation, which is created using the OpenGL API[12]. OpenGL renders primitives, which is a collection of vertices, that are processed through a series of programmable shaders. In the vertex shader each vertex is transformed through matrix multiplication with the homogenous vector $(x, y, z, 1)^T$, where (x, y, z) is the position of the vertex being transformed. The result of this is passed to the fragment shader, which calculates the colour of each fragment. Each These calculations are part of OpenGL's shader programs, which are executed on the GPU. This means that its speed is independent of the algorithm chosen for matrix multiplication of the CPU.

The matrix used to transform each vertex is known as the Model-View-Projection (MVP) matrix, which are the separate parts that the transformation can be split into. They are multiplied together in the following way

$$MVP = Projection \times View \times Model \tag{1}$$

The order of the operations is important, as the transformations are applied from right to left when using matrix multiplication. In this project an additional transformation matrix is added to the pipeline which is controlled by the user, giving

$$MVP = Projection \times View \times Transform \times Model \tag{2}$$

where $Transform$ is the transformation defined by the user.

The matrix that defines perspective projection is

$$\begin{bmatrix} \frac{1}{a*\tan\frac{fov}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3}$$

where fov is the field of view of the camera, a is the aspect ratio of the window, and far and near define the near and far plane of the veiw frustum. The matrix transforms this frustum into a cube with normalized coordinates in [-1, 1]. This is the range of coordinates in which most graphical APIs will render a vertex without culling it, including OpenGL which this project used. Figure 2 demonstrates the effect of this matrix on the view frustum and on a cube that lies within it.

The view matrix's purpose is to translate objects in the world so that they are in front of the camera, so it is calculated as $View = Orientation \times Translation$, where translation handles the camera's position, and orientation handles its rotation.

$$Translation = \begin{bmatrix} 1 & 0 & 0 & Pos.x \\ 0 & 1 & 0 & Pos.y \\ 0 & 0 & 1 & Pos.z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

where Pos is the position of the camera.

The orientation matrix calculation requires the local axis of the camera which are

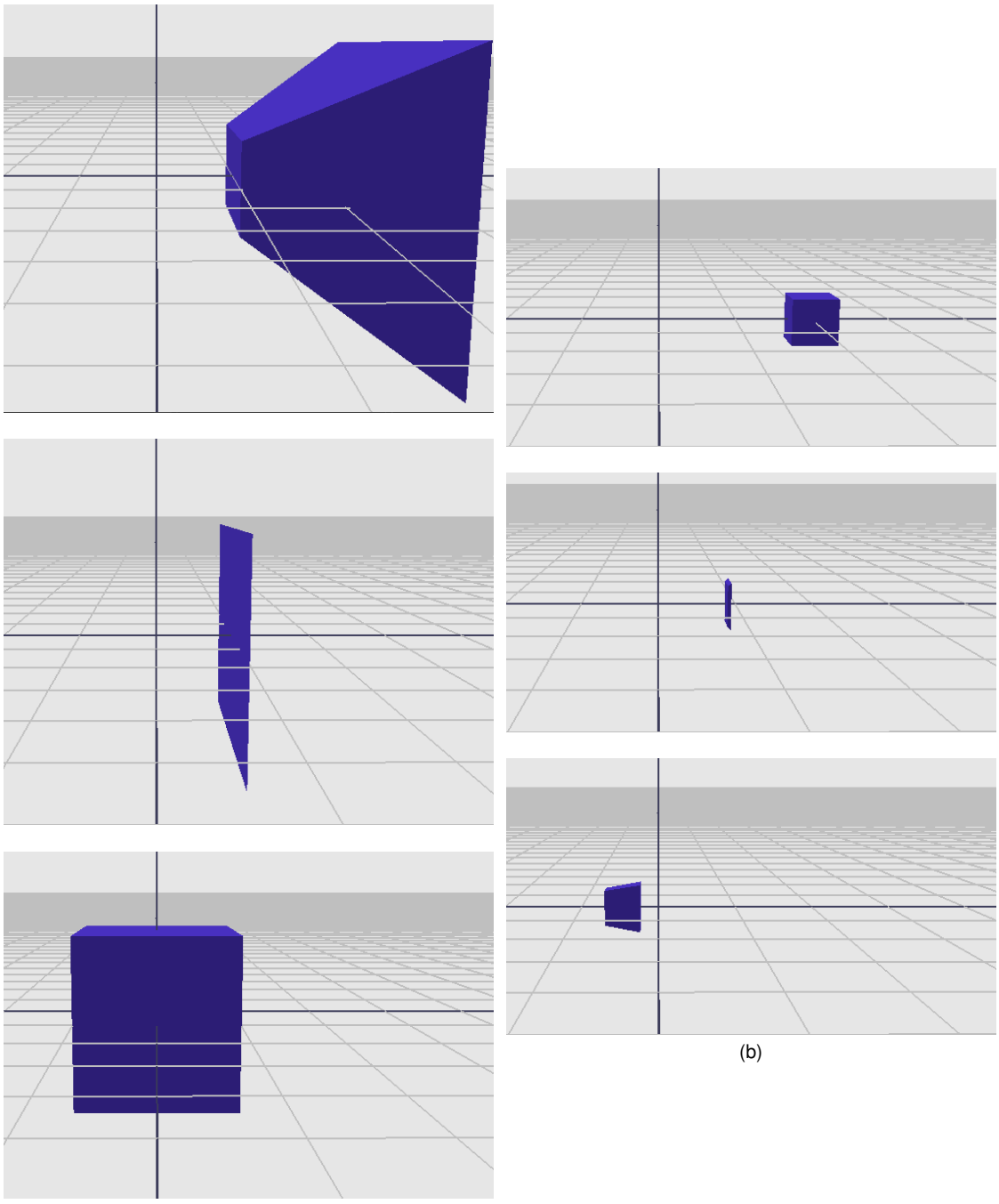
$$Z = |Target - Pos| \tag{5}$$
$$X = |LocalZ \times CameraUp| \tag{6}$$
$$Y = |LocalX \times LocalZ| \tag{7}$$

From here the orientation Matrix can be calculated as

$$Orientation = \begin{bmatrix} X.x & X.y & X.z & 0 \\ Y.x & Y.y & Y.z & 0 \\ Z.x & Z.y & Z.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

$$\implies View = \begin{bmatrix} X.x & X.y & X.z & X \cdot pos \\ Y.x & Y.y & Y.z & Y \cdot pos \\ Z.x & Z.y & Z.z & Z \cdot pos \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

The model and transform matrices could both be controlled by the user, allowing them to input expressions as each element of the matrix. This required that there was a scanner, capable of separating the expression into symbols, and a parser, which checked that the input was valid and created an expression tree from it. The scanning was simple, since most symbols in a mathematical expression are single characters and there are few types of symbol to check for. Scanners used in compilers construct finite state automata, which are then progressed through using the next character in the input string as the input to the current state. This ensures that the process is O(n), but for this project's purpose this is unnecessary due to the limited size of input strings.



(a) (b)

Fig. 2. (a) shows the initial view frustum which, was generated by applying the inverse of the perspective projection matrix to a cube, followed by the frustum midway through the perspective projection then the fully projected frustum. By generating the frustum in this way it demonstrates why the projection has this effect, since $A \times A^{-1} = I$ for any matrix A. (b) shows the effect of this projection on a cube that lies within the view frustum, which leads to it lying within the coordinates [-1, 1] in each axis.

2.3 Expression Parsing

Expressions are provided to the program as a string, so they must be parsed into an expression tree in order to be evaluated. This is done using a family of parsers called LR parsers, which stands for Left-to-right, Rightmost derivation in reverse. This means that strings are read from left to right without backtracking, and when there are multiple non terminal tokens to rewrite the rightmost token is always rewritten first. These properties mean that they operate in linear time.

An ideal parsing algorithm for mathematical expressions is the LR(1) parser, because it unambiguous, so for each expression there is only one possible tree [12]. As a result, simplification of the tree into a form that can be evaluated is much simpler. More importantly, mathematical expressions can be defined as an LR(1) grammar, which means that it can be parsed by an LR(1) parser. An LR(k) grammar is one that if the expression is scanned from left to right, then the correct reduction of a symbol can be made by looking at most k characters ahead [13].

LR parsers are a family of bottom up parsers [14] which means that it starts by creating a node for each token that is received by the scanner, then layers of parent nodes are added to create a tree that satisfies the grammar provided to it. A common implementation of it is a table-driven approach, which uses tables called the action table and goto table to simulate an automaton. The goto table stores which state to go to next for each input in a given state, and the action table stores information about how the expression tree should be modified for each input.

For these tables to be built, a model of the grammar is required that contains every state that the parser can reach, and the valid inputs in each of these states. This model is called the canonical collection of LR(1) items. An LR(1) item is a structure of the form $[A \rightarrow B \bullet C, a]$ that encodes information about a potential reduction. B represents part of the expressions that has been read, C is the part that is yet to be read, and a is the look-ahead symbol. If the look-ahead symbol a indicates that C is a valid continuation of the expression, then the placeholder symbol, $\bullet$, is moved forward, creating a new valid state for the parser. By repeating this process with different look-ahead symbols the canonical collection of the grammar can be built.

Since this structure requires a set of production rules for each non terminal symbol, using Backus-Naur form is the ideal method for defining the grammar used by the parser.

3 Methodology

The program was split into three major problems: handling of expressions, computation of linear algebra operations, and graphical display. This section will first outline the tools used for the project, and then will discuss each of these problems in turn.

3.1 Language Choice

C++ has many features that make it ideal for this project. Its operator overloading for classes means that an expression class can be created that acts similarly to floats and integers. This means that once matrices containing any one of these types were created, they can easily be modified to contain either of the others.

The language also contains SIMD intrinsic functions, which are compiler specific functions that allow for simple operations to be performed on multiple pieces of data at the same time. This improves the efficiency of dot products and matrix multiplication when the underlying type is a float. Also useful were the smart pointers, std::shared_ptr and std::unique_ptr, introduced in C++11, which were very useful in safely modifying the expression trees.

3.2 Graphics API

One of the most popular cross platform graphics APIs is OpenGL by the Khronos group, which has been used in games such as Doom(2016), and in popular software such as Blender and Google Earth, among many others. It renders shapes by accepting vertex data defining primitives, then transforming these primitives in shader programs, usually using matrices. This clear separation of the role of matrices made it ideal for the project, as this meant that the matrices defined by the user just had to be converted from expressions to floats then passed to the shader program to apply the transformation.

Khronos have also released Vulkan, which is a more low level API, aiming to give the developer more control. However, this comes at the expence of massively increased complexity. Since the program only displays simple shapes without lighting the main area to increase performance is in the calculation of the transformations, which is independent of the graphics API. As such the control of Vulkan is not required, so I will use OpenGL instead.

3.3 Expression handling

The aim of the expression handling was to be able to create an expression tree from each expression input by the user that was capable of evaluating itself. In this way it could be easily combined with the maths library by using it as the underlying type for matrices and vectors. This itself could be split into several sub-problems, which were expression scanning, parsing, and evaluation.

Expression scanning is the process of turning an input string into a list of tokens, which each have an assigned meaning. In this case it was made simpler by the knowledge that most tokens were represented by single character inputs. This meant that the scanner could maintain a pointer to a character in the input string, create a token from this character, then increment the pointer. In cases where further characters were required to identify the token, each possible multi-character token was compared to the relevant substring of the input. The number of multi-character tokens was small enough that this was suitably fast. The alternative method was to construct a finite state automota, which would ensure an O(n) run-time, but would also add increase the program's startup time as the automota is being created.

Also performed in the scanning was some basic modification of the expression based on common formats used in mathematics. For example, if there were multiple characters in a row that were not part of a function name, then multiplication operators would be added in between each character, and each one would be turned into a variable.

A table driven approach was used in the expression parsing, which meant that the goto and action tables had to be created from a grammar given in Backus-Naur form. The grammar for used to parse mathematical expressions is given in figure 3.

The creation of the action and goto tables required the canonical collection of the grammar. This was made by starting with the augmented production rule $[Goal \rightarrow \bullet Expression, eof]$, where eof is the end of file symbol. The canonical collection could then start to be obtained by calculating the closure of the augmented rule. Each LR(1) item was stored as an $std :: vector$ of enums, where the possible values were each terminal or non terminal symbol, or an arrow, or the placeholder symbol.

The closure uses the $First(\delta a)$ function, which returns a set containing every terminal symbol that can be the first symbol in δa, which obviously contains every element of $First(\delta)$, but it can also contain a if $First(\delta)$ contains the empty string ϵ. This function was implemented for a symbol a by recursively calling the $First$ function on the

⟨Goal⟩ ⊨ ⟨Expression⟩
⟨Expression⟩ ⊨ ⟨Expression⟩ + ⟨Term⟩
| ⟨Expression⟩ - ⟨Term⟩
| ⟨Term⟩
⟨Term⟩ ⊨ ⟨Term⟩ × ⟨Value⟩
| ⟨Term⟩ / ⟨Value⟩
| ⟨Value⟩
⟨Value⟩ ⊨ `Unary` ⟨Factor⟩
| ⟨Factor⟩
⟨Factor⟩ ⊨ (⟨Expression⟩)
| `Name`

Fig. 3. The grammar for mathematical expressions in Backus-Naur form[2]

while s has changed **do**
  for all $[A \to B \bullet C\delta, a] \in s$ **do**
    for all $C \to \gamma \in P$ **do**
      for all $b \in First(\delta a)$ **do**
        $s \leftarrow s \cup [C \to \bullet\gamma, b]$
      end for
    end for
  end for
end while

Fig. 4. The closure algorithm, operating on a set s of LR(1) items, with a set of production rules P

$stack.push(s_0)$
$word \leftarrow getNextToken()$
while $true$ **do**
  $state \leftarrow stack.top()$
  if Action[state][word] = $rA \to B$ **then**
    for $i \leftarrow 1; i \leq 2 \times |B|; i \leftarrow i + 1$ **do**
      $stack.pop()$
    end for
    $state \leftarrow stack.top()$
    $stack.push(A)$
    $stack.push(Goto[state][A]$
  else if $Action[state][word] =$ "si" **then**
    $stack.push(word)$
    $stack.push($"s_i"$)$
    $word \leftarrow getNextToken()$
  else if Action[state][word] = "acc" **then**
    return success
  else
    return fail
  end if
end while

Fig. 5. The LR(1) parsing algorithm: "si" means a shift instruction to state i, $rA \to B$ is a reduction from B to A, and "acc" is the acceptance of the expression.

first element of the right hand side of each production rule for a. This continued until a terminal symbol was reached, in which case the function would add the terminal symbol to a set of first symbols. Once each production rule had been checked, the set would be returned.

In order to calculate the closure of a set s, for each augmented rule $[A \to B \bullet C\delta, a] \in s$ any valid reductions following the placeholder symbol must be identified. This is done by adding the augmented version of each production rule for C, with the lookup symbol being each symbol that can follow C in the original rule. This new rule was then added to the set s. This process was then repeated until s is no longer modified by the process.

The final algorithm needed to compute the canonical collection was the goto algorithm, which simulated what happened to an augmented rule as the parser updated, which was modelled as moving the placeholder symbol forwards. For a symbol x, $goto(s, x)$ iterated over each augmented rule in s, and moved the placeholder symbol forwards if x had immediately followed it.

To start the canonical collection the closure of the initial augmented rule $[Goal \to \bullet Expression, eof]$ was computed. This closure was defined to be CC_0, and was used to create the set $CC = CC_0$. For every symbol x following a placeholder in CC_0, $goto(CC_0, x)$ was calculated. If the set produced was not already part of CC then it was added.

Each subset $CC_i \in CC$ that had not been processed then repeated the same process until CC was no longer being modified, and every $CC_i \in CC$ had been processed. This gave the final canonical collection CC. Once CC had been obtained the tables could be built by iterating over each augmented rule in the CC_i, and adding a shift, reduce, or accept action to the action table based on its form. The goto table was built by iterating over each non terminal n and calculating $goto(CC_i, n)$. if this gave an existing element of CC, CC_j, then $goto[i][n] = j$.

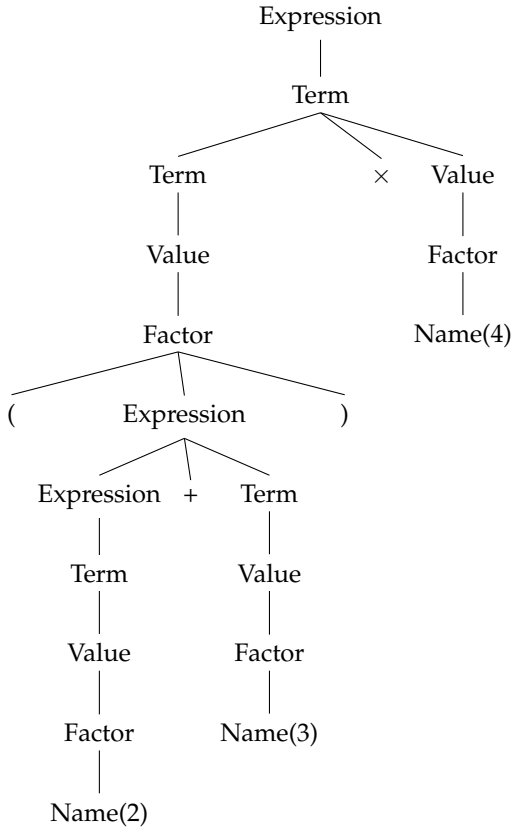
The output of the LR(1) parser was an expression tree with a parent expression node, with children corresponding to its BNF definition. In order to allow for efficient evaluation of the expression the tree had to be rearranged into a form such that each node is either an operation or a name. In this way a polymorphic class structure could be used to easily evaluate the tree. This structure consisted of a base Expression class that defined an evaluate function that took its child node as an argument. Each operator overloaded this function to return an operation on its child. Each type of name also overloaded it, but the child argument was not needed as they would always be a leaf node.

There were three main steps used in the simplification of the tree. The first of these was to remove brackets, which was done by modifying the bracket's parent so that its child was the first node after the open bracket. The node whose sibling was the close bracket also had to be modified so that the bracket nodes could be safely deleted.

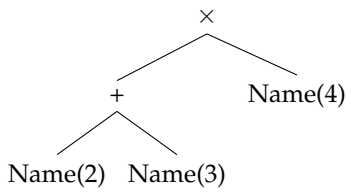
The next step was to modify operators so that instead of operating on the previous and next nodes, they instead operated on child nodes. This required modifying the parent node of the operator so that its first child was the operator, and modifying the operator so that its child was the parent's original first child. Various other modifications were also made to ensure that there were not loops in the tree structure.

The final step was to remove chains in the tree in order

Fig. 6. The expression tree generated on the input $"2 + 3) * 4"$ before and after simplification.



(a) The tree generated by the LR(1) parser



(b) The simplified form of the tree

to remove all instances of expression, term, value and factor nodes. The previous steps had ensured that each of these nodes would only have one child, so they could all simply be replaced by their child with no change to the tree's evaluation. The effect of these modifications can be seen on the tree created from the input $"(2 + 3) * 4"$ in figure 6.

Once the trees were simplified, the root node was stored in an expression class. In this way it allowed for mathematical operators to easily be overloaded for the class. Each overloaded function returned an expression whose root node was the operator being overloaded, and whose children was the root nodes of the operand expressions. The advantage of this method was that it worked for any number of operands, so the same technique could be used to apply functions to expressions such as applying trigonometric functions.

In this way expressions could be treated like base types of the language, the only difference being in their memory layout since the trees consisted of a root node which contained pointers to a child node, and to a sibling node.

A major benefit of storing expressions in this way, as opposed to evaluating its value immediately, was that variables could be used in them and have their value modified each frame without reevaluating the expression. To do this each variable class contained a single character and a static map from characters to floats. This map could be used to store the value of each variable in one place. By iterating over the elements of this map and creating a Dear ImGUI input using a reference to the value, the user could modify each variable.

3.4 Linear Algebra Operations

The basis of the linear algebra computation in this program is the matrix and vector data types. They each had a corresponding class for each dimension that was relevant in the project, and each matrix had a version where the underlying data type is a float, and one where it is an expression. The reason for this was that calculations that required no user input, such as the computation of the view matrix, had no reason to make use of expressions instead of floats. This was unnecessarily expensive when a calculation is repeated every frame since every operation involving expressions required the creation of a new expression tree. The other benefit of using floats where possible was that it allowed for the use of SIMD optimisation.

When using C++, SIMD processing was most easily implemented through the use of intrinsic functions designed to operate on vectors of floating point numbers. This project required up to four floats to be operated on, which for MSVC, the compiler used in this project, could be stored in an $\_\_m128$ data type. The most commonly used function was $\_mm\_mul\_ps$, which was used to multiply two $__m128$s component by component. This was used to calculate the dot product of vectors, and to calculate an element of the result of a matrix multiplication. The slowest part of these calculations was the creation of the $__m128$s from four floats, so floating point vectors stored them instead of the individual float values.

The most common linear algebra operator used in the project was matrix multiplication since it was required repeatedly to transform each vertex when rendering. Most of these calculations take place in the vertex shader, when the MVP matrix is multiplied by the homogeneous coordinates representing each vertex. In these cases I could not improve the efficiency of the multiplication itself, since it was already optimised by Khronos, but the number of calculations could be reduced by calculating the MVP matrix outside of the shader. This meant that it only had to be calculated once per frame, instead of once per vertex.

Each matrix also had to be converted into a form that could be passed to an OpenGL shader, which took a reference to an array of floats. This meant that the matrices had to evaluate each of their elements, and store the result in an array. A reference to this array could then be passed to the shader as a uniform.

For linear transformations the eigenvectors of the transformation matrix had to be calculated. This was not required for affine transformations so eigenvectors did not have to be calculated for four-by-four matrices, since these were only used to represent affine transformations in three

$$
\begin{aligned}
&vertices[] = \{ \\
&\quad Pos1_X, Pos1_Y, Pos1_Z, \\
&\quad Normal1_X, Normal1_Y, Normal1_Z, \\
&\quad Pos2_X, \ldots \\
&\}
\end{aligned}
$$

Fig. 7. The format of the floating point data used to define a primitive in OpenGL

dimensions. Because of this, it was not necessary to use the QR algorithm to calculate the eigenvectors. Instead, the equation $det(A - \lambda I) = 0$ was solved using the relevant polynomial formula. The real solutions to these were added to a list of eigenvalues.

For each of these eigenvalues, the solutions to $(A - \lambda I)\mathbf{v} = 0$ were calculated using Gaussian Elimination. In order to reduce the potential numerical instability of the problem partial pivoting was used, which meant that rows of the augmented matrix were swapped to ensure that the pivot element had the highest magnitude in its column. Since the right hand side of the augmented matrix started as **0**, it was impossible for it to be modified by the process, so it was unnecessary to store it. As a result the elementary row operations only had to be applied to $(A - \lambda I)$ until it was in its row reduced form. From here the eigenvector could be worked out by determining the free variables and assigning them a value, then solving the resulting equations accordingly.

The inverse of the matrices was also calculated using Gauss Jordan Elimination. In this case a structure contained a copy of the matrix whose inverse was being found, which was the left hand side of an augmented matrix, and the identity matrix, which was the right hand side. Elementary row operations were applied to both of these matrices at the same time until the left hand side became the identity matrix. At this point the right hand side had become the inverse of the original matrix.

3.5 Visualisation

OpenGL, a graphical API from the Khronus group, was the basis of the visualisation. This used a rendering pipeline that consisted of the definition of the shapes to be rendered, followed by the transformation of each vertex, then finally the calculation of each pixel colour. The shapes, known as primitives in OpenGL, were defined using a vertex buffer, and index buffer, and a vertex array object (VAO). The vertex buffer was created using an array of floats which can be seen in figure 7

Each vertex was stored in this array only once, even if it was used to construct multiple triangles, because the index buffer stored how these vertices were used to create the primitive. The vertex array object defined the memory layout of the vertex buffer so that the data could be accessed in the vertex shader. In this case the information stored in the VAO was that there was an attribute that contained three floats at the start of the data, with there being 6 floats of memory between each instance of this attribute. The structure of the normal attribute definition can be seen in figure 8.

$$
\begin{aligned}
&vertices[] = \{ \\
&\quad \overbrace{-0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f,}^{stride \text{ is 6 floats}} \\
&\quad 0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f, \\
&\quad \underbrace{0.5f, 0.5f, -0.5f,}_{offset \text{ is 3 floats}} \ \underbrace{0.0f, 0.0f, -1.0f}_{attributeSize \text{ is 3 floats}}, \\
&\quad \ldots \\
&\}
\end{aligned}
$$

bindVAO()
setAttribute(shaderLocation, attributeSize, stride, offset)

Fig. 8. The definition of the normal attribute. The $shaderLocation$ is the number used to refer to the attribute in the vertex shader. The $stride$ is the distance in memory between the start of each normal attribute, and $offset$ is the distance in memory from the start of the vertex data to the first normal attribute. The definition of the position attribute only differed in the offset and shaderLocation.

The normals were needed so that basic lighting calculations were applied to 3D shapes. This allowed for edges in the shape to be easily visible to the user so that they could properly understand the transformations that were being applied. The lighting was modelled as being directional with no attenuation, so its effect could be calculated as $factor = max(n \cdot dir, 0)$ where n was the normal vector and dir was the direction of the light. The normal had to be rotated according to the model and transform matrices without being translated so that the lighting remained accurate during transformations. In order to discount the effect due to the translation the normal was only multiplied by the upper left 3-by-3 sub matrix of $T \times M$, where T and M are the user provided translation and model matrices respectively. An additional ambient light modification was added to this result so that sides facing away from the light were not black, then this value was multiplied by the shape's colour to apply the lighting. Since this method guaranteed flat shading across each face, this could be done in the vertex shader instead of the fragment shader without any loss in quality. This was much faster since the fragment shader was called for each pixel as opposed to each vertex.

The clarity of the transformations was also improved by the camera controls available to the user, which were all controlled by the mouse. The mouse input was detected using GLFW, which allowed for me to define a callback function that was triggered anytime that the user moved the mouse. In this function I called $onMouseMove(Vector2\ newMousePosition)$ which was defined as a method for the camera controller.

In the case of the 2 dimensional transformations the user could drag the screen around the xy plane, which was done by offsetting the camera's position based on the change in the mouse's positions, provided that the right button was held down. For 3 dimensional transformations the camera revolved around the origin of the scene. This was done by storing the pitch and yaw of the camera's position from the positive z direction, and updating them based on the change in the mouse's position. Each frame the position of the camera was then calculated using the rotation matrices for pitching and yawing as shown in figure 9. This method also allowed for the camera to easily zoom in and out by changing the distance when the mouse wheel

$$position = Y \times P \times (0, 0, distance)^T$$

where the yaw matrix,

$$Y = \begin{bmatrix} cos(yaw) & 0 & -sin(yaw) \\ 0 & 1 & 0 \\ sin(yaw) & 0 & cos(yaw) \end{bmatrix}$$

the pitch matrix,

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(pitch) & sin(pitch) \\ 0 & -sin(pitch) & cos(pitch) \end{bmatrix}$$

and $distance$ is the distance of the camera from the origin

Fig. 9. The calculation of the camera's position for three dimensional transformations

was scrolled. This was implemented similarly to detecting mouse movement, with a GLFW callback function.

The final feature used to clarify the effect of the transformation matrix was the ability to animate the transformation. This was implemented by linearly interpolating between the identity matrix and the transformation matrix over a given time frame. By using the interpolated matrix as the transformation matrix in the shader, this created a smooth transition between the original and transformed shape.

Both of these camera types returned a four-by-four lookat matrix based on its position and orientation, so by using an abstract camera parent class these two camera types could be easily swapped when the user changed view types without changing how the lookat matrix was accessed. This meant that the only change to the shaders when the number of dimensions changed was the size of the transformation matrix. This was handled by creating multiple shaders at the start of the program for each transformation matrix size. By maintaining a pointer to one of these shaders, and changing which shaders it was pointing to when the user changed their view, this pointer could be used to bind the relevant shader, and set its uniform values accordingly.

3.6 User Interface

The user interface was created using DearIMGUI, which is an open source header only library created by Ocurnut. As the name suggests, the library used immediate mode rendering, so it was used by defining each element of the user interface each frame, and checking for user input at the same time.

In order to accept expressions as elements of the transformation matrix grids of string inputs were used, but these inputs also had to be capable of displaying expressions that had been generated by the program so that users could easily modify them. To do this each symbol defined a string conversion that could be used to build a complete expression. Brackets were added to certain symbols to guarantee that their meaning was correct, such as addition symbols which would return $(a + b)$. These conversions were displayed to the user, then reparsed if they were modified. DearIMGUI also required that a reference to a string was passed as an argument, so these strings were stored as a variable each frame.

The user also had the option of generating certain matrices which relied on the conversion between strings and expressions. These matrices were generated by creating expressions containing a single variable, then creating the matrices using these expressions as arguments. For example, the perspective projection matrix could be generated using variables for the field of view angle, the near and far plane, and the aspect ratio. This also indicated the need for the user to control the model transform matrix, as this would allow them to apply the projection to a frustum, or another shape of their choosing. As a result the application of the matrix in graphics was made much clearer in this example.

4 Results

The results are mostly drawn from visual feedback due to the nature of the project. The accuracy of the transformation matrices, the accuracy of the eigenvectors, and the control given to the user over the visualisation will be observed in turn.

4.1 Matrix Transformation

The transformation matrices can be selected by the user to be two, three or four dimensional square matrices based on whether they are intended to be affine or linear, and two dimensional or three dimensional. The accuracy of the transformation can be easily observed for common matrices, such as those that perform translations or rotations. Each element of these matrices can contain variables, which are modified by the user using either sliders or a float input. Figure 10 shows the transformation represented by the 2x2 rotation matrix for the a radians, where a is set to 0.6.

For matrices where the effect of the transformation is less obvious the transformation could be tested by using the matrix as the model transform, then performing the inverse transform to the resulting shape. An example of this is the perspective projection transformation demonstrated earlier in figure 2.

4.2 Eigenvector Visualisation

The eigenvector computation can only be performed for linear transformations, since there are no vectors that are only multiplied by a scalar in a transformation where the origin is not preserved. For these linear transformations the eigenvectors can be optionally displayed in red, which is seen in figure 11. The accuracy of these eigenvectors was checked by evaluating each element of the eigenvector and printing its value to the console, then using external tools to calculate the eigenvector separately. If these values matched, then both computations were assumed to be correct. This was repeated for multiple matrices which were manually chosen and input as the transformation matrix, and for every matrix chosen the eigenvectors matched.

For the 2D camera view the ray representing the eigenvector was checked by converting each eigenvector (a, b) into the equation $y = ax/b$, then comparing this equation to the ray generated. The existence of the grid made this comparison accurate enough for this purpose. For the 3D camera view a similar process was used where the eigenvector (a, b, c) was converted into the equations $y = ax/b$

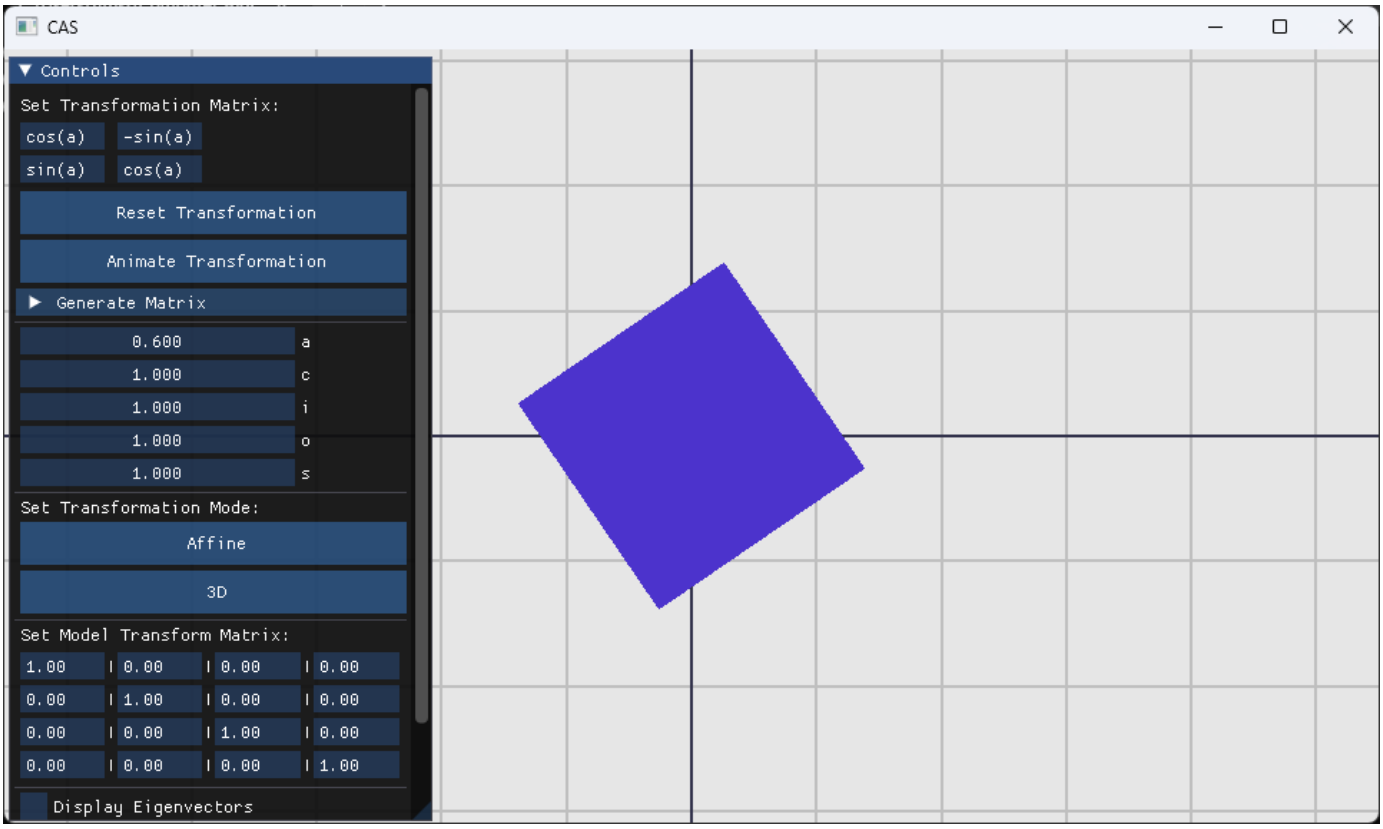


Fig. 10. The matrix provided is the 2 dimensional rotation matrix for an angle a radians, which has been set to $a = 0.6$.

and $z = ax/c$. Positioning the camera so that the x-z plane and the grid could be seen allowed for the second equation to be compared to the ray from above, and similarly the camera was positioned to view the x-y plane so that the ray could be compared to the first equation.

The final method used to evaluated the accuracy of the eigenvector visualisation was to animate the transformation applied by the matrix, and visually check that points lying on the eigenvector remained on it throughout the animation. For the 2D view applying the transformation to the grid made this easy, as for each matrix a point could be found where the corner of a grid square aligned with the eigenvector. This point could then be tracked to assert the correctness of the ray. This was not feasible for the 3D view as no clear identifiable intersections with the ray could be found for most matrices.

4.3 User Input

The view can be either two dimensional or three dimensional, and the camera controller changes accordingly. For the 2D view the user can pan the camera around the x-y plane, and move the camera forwards and backwards to zoom, but the view direction can not be changed. For the 3D view the camera's view target is fixed at the origin, and its position can be rotated about the origin and moved towards or away from it. The 2D view can be seen in figure 2 and the 3D view can be seen in figure 11.

The user interface gave the user control over the transformation and model matrices through grids of text inputs which could each receive an expression. Alternatively common matrices could be generated as the transformation matrix, which would create variables as required. For example when the perspective projection matrix was generated, variables would be created to control the aspect ratio, field of view, near plane and far plane of the view frustum.

The final set of features available to the user was the ability to control whether the matrix transformation was applied to the grid, whether the box was visible, and whether the eigenvectors were visible. The application of this is explored in the evaluation.

4.4 Performance

Also important to the user's understanding is the performance of the program, as they are unlikely to engage with the tool fully if there is any noticeable lagging or similar issues. When tested on my computer the program never exceeded 3ms per frame for the majority of the program's runtime. It only exceeded this while animations were playing, at which point it could reach up to 30ms, or on a frame where the eigenvectors were calculated, where it could reach up to 10ms.

5 Evaluation

This project is evaluated based on its potential for improving user understanding of linear algebra, which can be split into three main sections. These are how much the program's functionality can improve understanding when effectively used, how effectively the user interface guides the user towards this effective use, and whether the program is

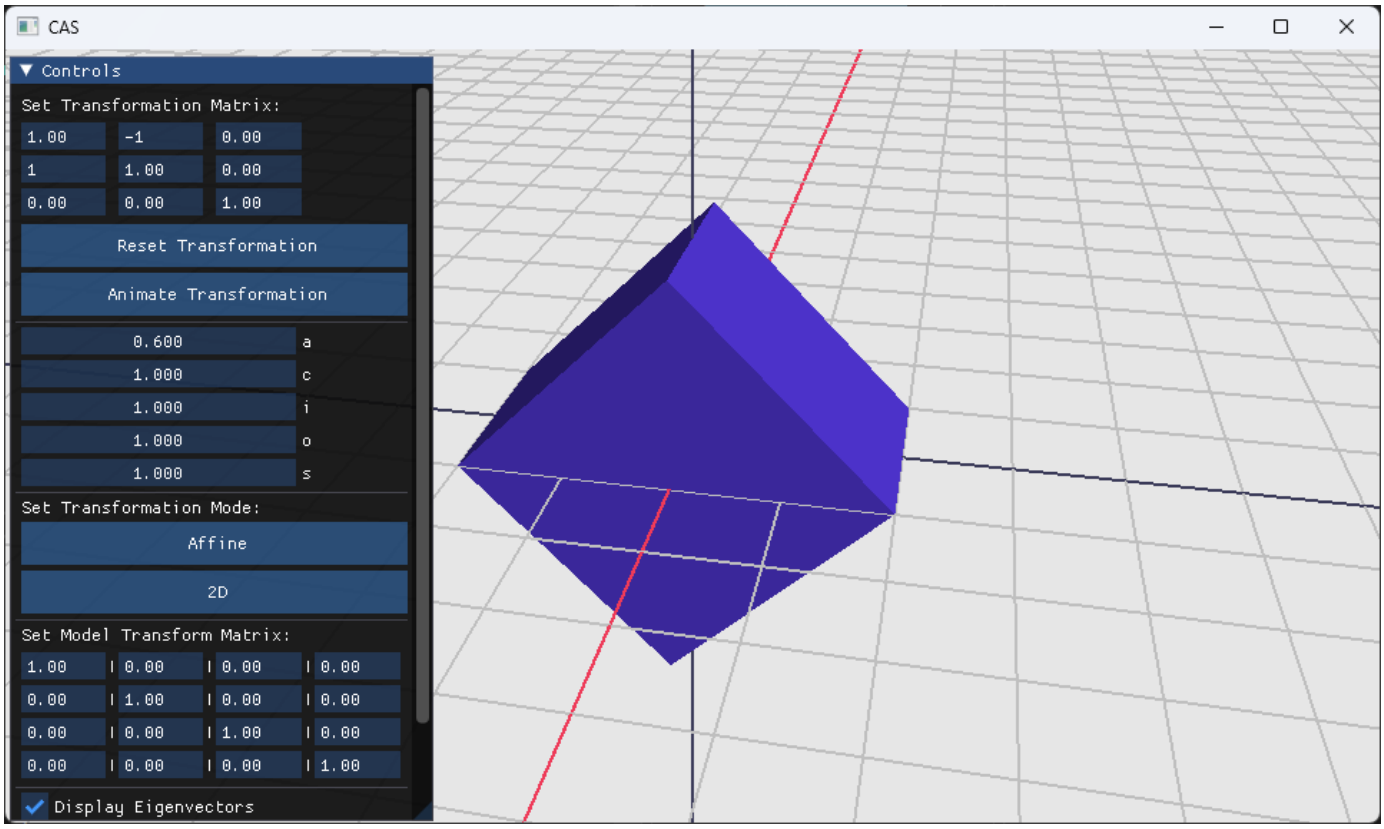


Fig. 11. The effect of the transformation is a rotation about the z axis, so the correctness of the eigenvector can be easily observed, since the z axis is obviously not transformed by this matrix.

implemented efficiently enough to avoid harming the user experience.

The usefulness of the program's functionality will be considered first, starting with the application of the two matrices available to the user. The basic functionality of the matrix transformations teaches the user about an application of linear algebra, since this is its main use in computer graphics. This can be demonstrated further by defining the transformation matrix to be the look-at matrix, or the perspective projection matrix which can be generated automatically. By animating these transformations the user can observe how 3D worlds are projected onto 2D screens. However, to make these observations the model transformation must be set to specific values, such as to generate the frustum to observe the projection matrix. A user unfamiliar with the topic would never do this, so they would not reap most of the benefits of this system. The ability to generate scenarios, which set both of the matrices to the required value, would solve this problem and in doing so would enhance the user's ability to understand linear algebra's application.

The matrix input also allows the user to understand the effect of certain components of a matrix when matrix multiplication is performed, such as how the final column in an affine transformation controls the translation of the shape. This is also something that the user may not notice on their own, but the ability to generate transformation matrices would make this clear to them. For example, if they chose to generate a scaling matrix, where the shape is scaled by a factor of sx, sy and sz in the x, y and z directions respectively, then the user would likely notice that sx, sy and sz formed the diagonals of the transformation matrix, prompting them to modify these diagonal values to understand their meaning.

The effectiveness of the eigenvector visualisation is examined next, which differs between the 2D and 3D view. For the 2D view the display of the eigenvectors is very clear, as a ray that is fixed on an x-y grid is very easy for a student studying linear algebra to interpret. If the grid is transformed by the matrix then the meaning of the eigenvector becomes apparent, as the grid scales in the direction of the eigenvector. The problem with this is that there is little encouragement for the user to do so, so they may try and understand their meaning from the transformation of the box alone, which for some matrices is not clear as there are no identifiable features lying on the eigenvector. For the 3D view there are similar problems, as it is hard to track points lying on the eigenvector unless it closely follows the x-z grid, so the only times where the eigenvector display is meaningful are when the ray intersects with an edge or corner of the box.

The animation of the matrix transformation makes all of the previous features much easier to understand because it allows the user to track the box throughout its transformation. This makes it easier for them to understand what transformation the matrix represents in some cases. For example, a rotation matrix of $\pi/2$ radians would not have any visible effect without animation, since the box has rotational symmetry, so by animating the process this problem can be avoided. It is equally useful for the eigenvector display

because it means that the user can track a point lying on the eigenvector continuously, instead of trying to convince themselves that a point stayed on the eigenvector from the start and end points alone.

The effectiveness of how the user is guided by the interface will be explored next. The control given to the user over each element of the program has been examined throughout this section already, so instead the user interface(UI) is evaluated based on how effectively it guides the user to these controls. This will be done using the principles of good interface and screen design given in [15]. These were created for the purpose of designing a web page so certain points will be ignored or emphasised where relevant. The first of these is that each element of the UI must serve a purpose, which is satisfied by the program since each input is labelled as simply as possible. Unnecessary labelling is also avoided by making the program's current state clear through other means, such as the transformation matrix size and camera controller indicating whether the view is 2D or 3D.

Another principle stated is that there is a clear starting point for the user's attention. In the program this starting point is the input for the transformation matrix, which is the most important aspect of the program. To draw attention to it, it is placed in the upper left quadrant, since this is the area that users first notice. In graphical displays it has been shown that text is noticed before images, larger font sizes before smaller, and changing information before static. This highlights an area of improvement for the UI, since the same font size is used throughout. However, dynamic information is used to draw the user's eye to the transformation matrices during their animation, since its value is constantly updated throughout.

The ordering and grouping of the data must follow a logical sequence, with groups that are organised based on the relationship between the data. This is satisfied by the UI, since there are clear groups containing data about the transformation matrix, the variables, the type of transformation, the model transform, and finally the display settings. These are ordered in the program as written, which is the order of importance to the user's experience. In this way the user is guided to focus on the program's core features first. This layout also benefits the program's navigation and flow, which is also important to the UI's design. The aim is to provide a natural progression of the user's focus throughout the UI in a top-to-bottom, left-to-right manner. This can be helped by aligning elements and ordering them so that the user's eye movements are reduced. For example, once the user inputs a variable into the transformation matrix a slider appears directly below to set its value, which minimises the distance that the user has to move their focus.

Also important is how visually pleasing the composition of the UI is, which in this project is achieved through the use of symmetry and regularity. The buttons are positioned in the centre of the windows, which makes the space feel symmetrical due to the one column layout, and the spacing between each element is consistent. The grouping of elements is made more visually appealing though the use of subtle lines dividing them, so their separation is clear without adding visual clutter. A drawback of this approach is that each element is close together because of the lack of white space, which may be too busy for the user. For a new user, this could make the UI harder to understand without instruction.

The project fails to use focus and emphasis effectively in its UI since the lightest elements, which are the ones most likely to draw the user's attention, are the buttons. The matrix inputs are the core feature, so they should be lightest element to highlight their importance to the user. No other elements have features that create emphasis, so the UI appears dull.

Finally the performance of the program will be evaluated. Although it ran with no issues during testing, any potential performance issues must be identified so ensure that the program is also effectively when used on weaker machines. The two areas for concern identified were the animations and the calculation of the eigenvectors. For the eigenvectors, reducing the time taken any further is not feasible due to the complexity of the operation, so instead it may be suitable to perform this calculation in a separate thread. This would not reduce the time taken for the eigenvectors to be calculated, but it would mean that the user input is still detected during its calculation which is key to reducing the perception of lagging.

The animations have more room for improvement, as every frame a new expression tree is created for each element of the matrix that is modified from the identity matrix. This is unnecessarily complex, as the user does not receive any benefit from the matrix being kept in expression form throughout this process. For this reason the program would benefit from converting the transformation matrix to floating point values before the animation takes place. It could also benefit from updating multiple values in the matrix at once through the use of SIMD intrinsic functions, although this is not guaranteed to lead to a performance increase.

6 Conclusion

The aim of this project was to produce a tool that could be used to visualise linear algebra operations, with the aim of helping students in their understanding of the topic. The requirements for the tool were that 2D or 3D shapes could be transformed by matrices representing linear or affine transformations, where each element was a mathematical expression, and for the computation and visualisation of the eigenvectors of these matrices to be possible.

These requirements were completed to an extent such that with proper guidance this project could be used as tool to improve user understanding of matrix multiplication and eigenvectors when the 2D view is used, as the grid gives the user clarity over what they are seeing. However, for the 3D view it is harder to understand the eigenvector visualisation fully because there are not enough reference points to understand the orientation of the rays, or their meaning. Only certain matrices have eigenvectors that lie on the x-z grid, and for these vectors their meaning can be interpreted similarly to the 2D view, but for any other vector the user has no way to accurately interpret its value. This could be improved upon by providing the user with the exact value of the eigenvectors when they are calculated, or allowing the user to change the position and orientation of the grid.

The accuracy of the expression parsing is a success for this project, as it allows the user to input expressions in a similar manner to as if they were writing them manually, such as not requiring a multiplication symbol between variables, or removing brackets from common functions such as writing $\cos \pi$ instead of $\cos(\pi)$. Features such as this and the automatic generation of variables when they were referenced made inputting an expression very simple for the user, which is vital for an educational tool.

In order for this program to be of use for educational purposes it was important that it was efficient enough to run smoothly even on weaker computers, as students are unlikely to have access to a powerful computer when first learning about linear algebra. This requires further testing on other computers with weaker hardware to understand whether this was fully achieved, but the results of the testing performed on a higher end computer indicate that it was, since the frame time of 3ms is much smaller than is required for a smooth experience for the user. The downside of this program's implementation is that the computer must have a GPU to run the program, as OpenGL performs the shader programs on it. This is unavoidable because the calculations performed on the GPU are far too numerous to be performed on the CPU in any reasonable amount of time.

There are several areas in which the project could be extended, which are predominantly in the evaluation of expressions. In the current program expressions can either be evaluated as floats, or converted to strings. These string conversions operate by outputting every operator that makes up the expression without simplification, so for expressions that have had several operations performed on them the resulting string is unnecessarily complex, often to an extent that it could confuse the user. This is damaging to the project's main goal of improving the user's understanding. Implementing this functionality would allow the project to be extended so that user feedback could be provided where there currently is none, such as by displaying a matrix's eigenvectors in exact form. This would also remove the issue of numerical instability from the program because each expression could be kept in exact form then simplified, instead of attempting to evaluate part of the expression too early.

The other main improvement that could be made is in the UI design, which currently uses the default layout and design of Dear ImGUI. This design is ideal for testing the functionality of the program, but it fails to clearly indicate how the project should be used to its full capability without external guidance. For the program's current state this is not too much of an issue, since its feature set is relatively limited, but the UI's current design would be hard to understand if it grew in size.The main improvements required are to use colour to highlight the important features of the program, which would also serve to make the UI more visually appealing.

Despite these issues the program still provides a clear visualisation of matrix multiplication for the 2D and 3D views, and an equally clear representation of eigenvectors for the 2D view. The only failing of the project is in the clarity of its representation of 3D eigenvectors, which doesn't take away from the understanding gained in the 2D view. For this reason the project successfully achieves its aim of improving the user's understanding of linear algebra operations.

REFERENCES

[1] L. Ermann, A. D. Chepelianskii, and D. L. Shepelyansky, "Toward two-dimensional search engines," *Journal of Physics A: Mathematical and Theoretical*, vol. 45, no. 27, p. 275101, jun 2012. [Online]. Available: https://dx.doi.org/10.1088/1751-8113/45/27/275101

[2] (2020) Opengl mathematics. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html

[3] (2021) Eigen. [Online]. Available: https://eigen.tuxfamily.org/index.php?title=Main$_{P}age$

[4] 3blue1brown. [Online]. Available: https://www.3blue1brown.com/

[5] Desmos. [Online]. Available: https://www.desmos.com/

[6] Mathematica. [Online]. Available: https://www.wolfram.com/mathematica/

[7] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, p. 354–356, 1969.

[8] D. Coppersmith and S. Winograd, "On the asymptotic complexity of matrix multiplication," in *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, 1981, pp. 82–90.

[9] S. Pradyumna, "Performance comparison of matrix multiplication algorithms," in *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, 2017, pp. 461–466.

[10] J. W. Demmel, *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611971446

[11] N. Gould, "On growth in gaussian elimination with complete pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 12, no. 2, pp. 354–361, 1991.

[12] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*. USA: Prentice-Hall, Inc., 1972.

[13] D. E. Knuth, "On the translation of languages from left to right," *Information and Control*, vol. 8, no. 6, pp. 607–639, 1965. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995865904262

[14] K. D. Cooper and L. Torczon, "Chapter 3 - parsers," in *Engineering a Compiler (Third Edition)*, third edition ed., K. D. Cooper and L. Torczon, Eds. Philadelphia: Morgan Kaufmann, 2023, pp. 85–157. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128154120000097

[15] W. O. Galitz, *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. USA: John Wiley amp; Sons, Inc., 2007.