PERSPEX MACHINE V: COMPILATION OF C PROGRAMS

Perspex Machine V: Compilation of C Programs

Matthew P. Spanner & James A.D.W. Anderson[*]
Computer Science, The University of Reading, England

Abstract

The perspex machine arose from the unification of the Turing machine with projective geometry. The original, constructive proof used four special, perspective transformations to implement the Turing machine in projective geometry. These four transformations are now generalised and applied in a compiler, implemented in Pop11, that converts a subset of the C programming language into perspexes. This is interesting both from a geometrical and a computational point of view. Geometrically, it is interesting that program source can be converted automatically to a sequence of perspective transformations and conditional jumps, though we find that the product of homogeneous transformations with normalisation can be non-associative. Computationally, it is interesting that program source can be compiled for a Reduced Instruction Set Computer (RISC), the perspex machine, that is a Single Instruction, Zero Exception (SIZE) computer.

Keywords: perspex machine, compilation, RISC, SIZE.

1. Introduction

The perspex machine is the ultimate Reduced Instruction Set Computer (RISC). It is a Single Instruction, Zero Exception (SIZE) computer. This makes it a good target for a compiler and, we suppose, makes it easy to implement in a massively parallel architecture, though we explore only serial computation here. The machine also derives interesting properties from its geometrical nature.[7] In particular, it can be implemented as a sequence of perspective transformations and conditional jumps.[4] This might lead to fast implementation as an optical computer.[4]

The perspex machine was introduced[4] by unifying the Turing machine with projective geometry. The halting perspex, H, was later re-specified.[5] The perspex machine operates on transreal numbers[3,5] that support a total arithmetic. The number nullity, $\Phi = 0/0$, is both transrational[3] and transreal,[5] as is the number infinity, $\infty = 1/0$. The transreal numbers are the union of the strictly transreal numbers, $\{\Phi, \infty\}$, with the real numbers. Nullity lies off the real-number line and infinity lies at its positive extreme. The strictly transreal numbers occur as co-ordinates in perspex space[4] and, amongst other things, ensure that the Turing *halt* is a discontinuous operation, despite continuity over all other Turing operations.[6] Various forms of the perspex are illustrated in,[5,10] though we are concerned only with the matrix and computer instruction forms here.

The perspex can be a 4×4 matrix with successive column vectors x, y, z, and t, or it can be a computer instruction: $\vec{x}\vec{y} \rightarrow \vec{z}$; $\mathrm{jump}(\vec{z}_{11}, t)$. The perspex machine operates in a 4D space, called *perspex* or *program* space,[4] that contains perspexes at every point. The machine executes the perspex at a point as an instruction. The machine reads the perspexes at locations x and y, multiplies them together and writes the product,[4] reduced to canonical form,[5] into the location z. It then examines the top left element, $\vec{z}_{11}$, of the product and constructs a relative jump from the current location using the components of t. If $\vec{z}_{11} < 0$ it jumps by t_1 along the x-axis, otherwise if $\vec{z}_{11} = 0$ it jumps by t_2 along the y-axis, otherwise if $\vec{z}_{11} > 0$ it jumps by t_3 along the z-axis. In every case it jumps by t_4 along the t axis. Thus, the machine starts at some point and control jumps from point to point until H is encountered.

[*] Corresponding author. author@bookofparagon.com, http://www.bookofparagon.com
Computer Science, The University of Reading, Reading, Berkshire, England, RG6 6AY.

Previous work used hand-written perspex programs executed on a digital computer using a simulation[4] of the perspex machine implemented in the AI language Pop11.[1] Here we describe a recursive descent compiler, implemented in Pop11, that reads C source code and compiles it to perspexes. The perspexes are visualised and executed in approximate simulations of the perspex machine implemented in both Pop11 [5] and C++. [10]

The compiler generally lays out perspexes along a straight, spacetime line that changes only in time, t. However, the compiler implements C conditionals by jumping additionally in the spatial dimensions x, y, or z. C loops exploit a backward jump in time to iterate the loop. This exposes an infelicity in the specification of the perspex machine. Whilst the perspex machine can support such arbitrary temporal jumps[4,5] it cannot copy them directly. It must use additional geometrical transformations and the access column[5] to prevent an arbitrary temporal jump component being reduced to a normalised jump component[5] of 0, 1, Φ, or ∞. This is remedied by the introduction of the universal perspex machine.[11]

We now list certain standard perspexes and describe how various parts of the C programming language are compiled to perspexes. We end with a discussion of the merits of the compilation templates used here and with suggestions for future work.

2. Standard Perspex Matrices

The matrices used by the compiler have at most one variable element with the rest of the matrix fixed. It is, therefore, generally convenient to write a compiler matrix as A_x where the matrix A has all elements fixed, except for the element that takes on the value x. A different interpretation is used for the subscript on the J and S matrices. The name of the matrix, A, B, or whatever, indicates the lay out of the matrix.

Many of the compiler matrices have their middle two rows set to zero. This prevents the introduction of cross-terms in the perspex multiplication and division functions. Whilst such cross terms are harmless, it is more elegant to prevent their occurrence by zeroing these two rows. In the following J matrix[2] the subscript, 9, is the binary code made by the pattern of zeros and ones read along the major diagonal from the least significant bit at top-left to the most significant bit at bottom-right. Premultiplication by J_9 zeros the middle two rows of a matrix.

$$J_9 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad \text{(Eqn 1)}$$

The S matrix is taken from[4] and a variant of it, S_a, is defined here. These matrices are used, amongst other things, to subtract terms during the computation of jumps. Note that $S_a = D_{-1}$ in (Eqn 10).

$$S = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}. \qquad \text{(Eqn 2)}$$

$$S_a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}. \qquad \text{(Eqn 3)}$$

The following two matrices are taken from:[4]

$$C = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \text{(Eqn 4)}$$

$$G = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \qquad \text{(Eqn 5)}$$

The following matrix is taken from:[5]

$$H = \begin{bmatrix} \Phi & \Phi & \Phi & \Phi \\ \Phi & \Phi & \Phi & \Phi \\ \Phi & \Phi & \Phi & \Phi \\ \Phi & \Phi & \Phi & \Phi \end{bmatrix}. \qquad \text{(Eqn 6)}$$

A C program needs representations for variables and numbers so that it can carry out arithmetical operations. The perspex representations for variables and numbers support all of the compiler templates and C structures used here. These representations are based on matrices occurring in the original unification.[4] V_x represents a variable with value x. V_x is based on G in (Eqn 5). In particular, $V_0 = G$.

$$V_x = \begin{bmatrix} x & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \qquad \text{(Eqn 7)}$$

The matrix C_x represents the number x. C_x is based on C in (Eqn 4). In particular, $C_1 = C$.

$$C_x = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad \text{(Eqn 8)}$$

Observe that the product $V_xG = C_x$ so that a variable is converted to a number by post multiplication with G. Note, too, that $G^{-1} = G$, whence $C_xG = V_x$. In other words, a number is converted to a variable by post multiplication with G, and *vice versa.* It would be interesting to know if all type polymorphisms in a useful computer language can be implemented by simple geometrical transformations. If so, formal proofs of the completeness of the type system, and of the correctness of the casting of types, would be simpler than at present. Type systems would then be subject to interpolation and approximation by filtering methods.[6]

The matrix M_x is a dilatation by the scale factor x. This matrix is used to obtain the arithmetical operation of multiplication.

$$M_x = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad \text{(Eqn 9)}$$

It would be natural to define a matrix D_x that, as a result of the normalisation step,[5] performs the arithmetical operation of division, but this approach was not taken in the compiler.

$$D_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & x \end{bmatrix}. \qquad \text{(Eqn 10)}$$

3. Storage of Numerical Types

The compiler identifies storage for standard perspexes in the $t = 1$ hyperplane of the compiled program. Conventionally, the $t = 0$ hyperplane is reserved for the internal structure of a perspex machine. Entering the machine in this hyperplane allows the machine to initialise itself before control passes to hyperplanes with $t > 0$. The universal perspex machine[11] conveniently supports backward jumps in time, so the $t = 0$ hyperplane may also be used for post-processing prior to a halt in that machine. The current machine stores C's *main* function in the $t = 0$ hyperplane, data in the $t = 1$ hyperplane, and subfunctions in the $t = 2$ hyperplane. Thus, it maintains separate data and instruction spaces.

The layout of the standard perspexes is shown in Figure 1:. The figure shows a left-handed, co-ordinate frame with axes labelled X, Y, and Z. This is for display purposes only. Perspex space is conventionally indexed as a right handed space. The matrix S is stored at $(3, 0, 0, 1)$, shown as s in Figure 1:. The matrix S_a is stored at $(3, 1, 0, 1)$ and is shown as sa. The matrix G is stored at $(1, 0, 0, 1)$ and is shown as g. The compiler stores C_x matrices in a column of matrices called the *Numerical Constant Column*. A matrix C_x is stored at $(2, x, 0, 1)$. The compiler always generates and stores C_{-1}, C_0, and C_1 for use in jump calculations. It also stores a C_x for each distinct, literal number x that it finds in the C source. In a continuous perspex machine C_x would be generated and stored for all transreal x, thereby creating an analogue of the augmented real-number line. The identity matrix $I = J_{15} = C_0$ is shown as i.

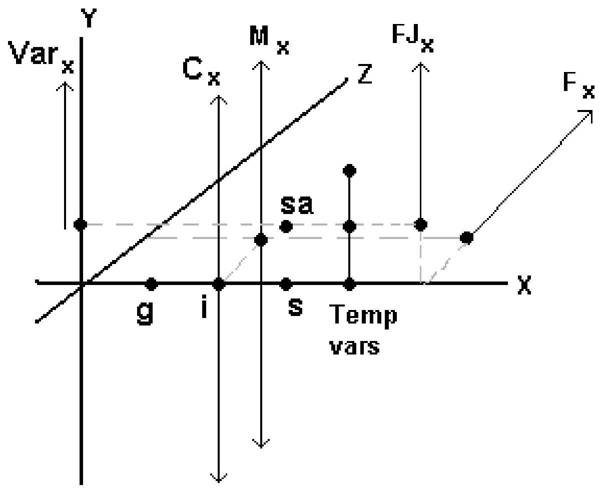

Figure 1: Standard perspexes in the $t = 1$ hyperplane.

The *Variable Column,* shown as Var_x, contains all of the global and local variables in the order the compiler finds them in the C source. The n^{th} variable found is stored as V_x at location $(0, n, 0, 1)$. Locations $(0, k, 0, 1)$, with k greater than the largest value of n, are used as a stack of temporary variables for use in evaluating sub-expressions. The compiler does not support recursive functions. In future, one might consider whether to store variables local to recursive functions in a stack, or whether to store them in columns running orthogonally to the Variable Column.

Three temporary variables are used as a scratch space, roughly analogous to an accumulator in a standard Central Processing Unit (CPU). The scratch locations are $(4, 0, 0, 1)$, $(4, 1, 0, 1)$, and $(4, 2, 0, 1)$. They are shown as *Temp vars.*

The *Multiplier Column,* M_x, is similar to the Variable Column. It stores matrices M_x at locations $(2, x, 1, 1)$. These matrices are used to effect multiplication and division by literal numbers. As before, the compiler generates matrices M_x only for the multiplications and divisions found in the C source, whereas a continuous perspex machine would generate and store all matrices M_x. The matrices M_x and V_x are not permutations of each other so there is no simple matrix that carries the one into the other. This prevents a simple implementation of the multiplication of a variable by a variable, or the division of a number or variable by a variable. These cases are handled by compiling multiplication and division functions from C source that performs multiplication and division iteratively. These functions operates to finite precision.

The *Function Output Column,* F, holds one V_x perspex for each function. The n^{th} function found in the C source uses the location $(5, 0, 1, n)$. This location is used to hold the return value of its corresponding function.

The *Function Jumper Column,* F_j, holds number perspexes, C_x, that are used to compute return jumps from a function call. The x^{th} lexical call to the n^{th} function in the C source puts C_x into the location $(5, n, 0, 1)$.

4. Jumps, Conditionals, and Loops

The jump part, $\mathrm{jump}(\grave{z}_{11}, t)$, of the perspex instruction examines the sign of z_{11} then jumps in the time axis and at most one spatial axis. If a jump along more than one spatial axis is wanted then successive jumps must be taken. The default jump for every instruction has the t column set to $\begin{bmatrix} 0 & 0 & 0 & k \end{bmatrix}^T$. This forces a jump of k along the t-axis. The t-columns $\begin{bmatrix} k & 0 & 0 & 0 \end{bmatrix}^T$, $\begin{bmatrix} 0 & k & 0 & 0 \end{bmatrix}^T$, $\begin{bmatrix} 0 & 0 & k & 0 \end{bmatrix}^T$ force a jump of k, respectively, along the *x-, y-,* and *z*-axis. The compiler can compile jumps with arbitrary values of k, but the value $k = 1$ is used by default. A perspex whose only function is to cause a jump is called a *jumper.* The read/write parts of a jumper, that is the *x-, y-,* and *z*-columns of a perspex, can be set to identity if the conventional allocation of the identity perspex machine is maintained.[4]

The compiler handles C conditionals of the form: *if (a relop b) c else d.* See Figure 2:. Here *a* and *b* are expressions that result in a number, *relop* is one of C's relational operators, and *c* and *d* are, possibly bracketed, expressions. The clause *else d* is optional. The conditional, *a relop b,* is evaluated by forming the subtraction *a - b* using the perspexes S and S_a, compare with,[4] and performing a $\mathrm{jump}(\grave{z}_{11}, t)$. If the C *relop* evaluates to *True* then a jump is taken in the time axis and the appropriate spatial axis as given by $\mathrm{jump}(\grave{z}_{11}, t)$. Alternatively, if the C *relop* evaluates to *False* then the jump is taken in time with a zero component in the spatial axes. If the *relop* has two satisfying conditions, such as in >= or <=, then the appropriate code is laid out in the equality, =, branch. The other branch jumps to it so that the code is not duplicated.

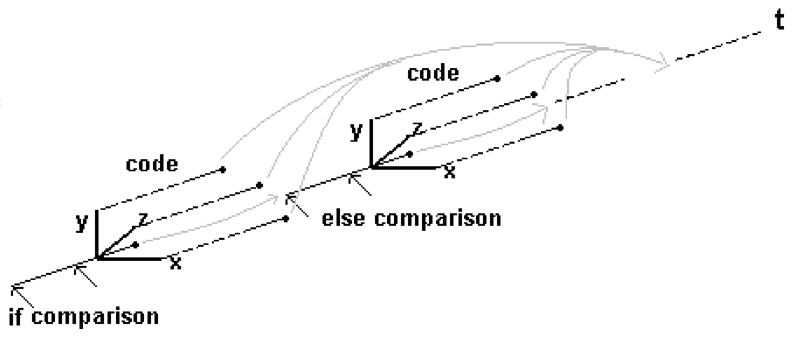

Figure 2: If-Else conditional statement.

The compiler handles C's *while* statement by evaluating a conditional and branching spatially if the condition is *True.* One spatial branch holds the code to be executed and the others, if any, hold jumps to this code. Alternatively, if the condition is *False* a jump is made only along the time axis, thereby leaving the loop. The compiler handles C's *for* loop in a similar manner.

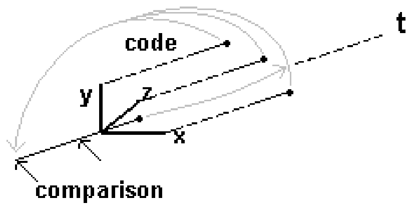

Figure 3: While loop.

5. Function Call and Return

In the C source, the x^{th} lexical call to the n^{th} function with arguments a_i is computed in a series of steps. Firstly, the arguments, a_i, are evaluated and stored as variables, V_{a_i}, in the Variable Column. See Figure 1:. As only serial, non-recursive functions are supported, there is a one-to-one correspondence between the formal arguments of a function, a_i, and the V_{a_i} during the life of the call. Secondly, C_x is written into the Function Jumper Column. The number x is used during the function return to compute a jump back to the end of the x^{th} lexical call of the function. Thirdly, a jump is made to the function. Fourthly, the function is evaluated and the return value is written into the Function Output Column for later use. Fifthly, a return to the x^{th} lexical call of the function is computed as follows. The number, C_x, in the Function Jumper Column, is decremented by one. If the result is non-zero then a jump is made along the *z*-axis and the process of decrementing C_x and jumping is repeated. Thus, control jumps x units along the *z*-axis. At this position C_x has been decremented to zero and a jump of one unit is made in the *y*-axis. This arrives at the first of a chain of jumpers that returns control to the end of the x^{th} lexical call to the function. Thus, a single block of code implements all of the calls to, and returns from, a serial, non-recursive function.

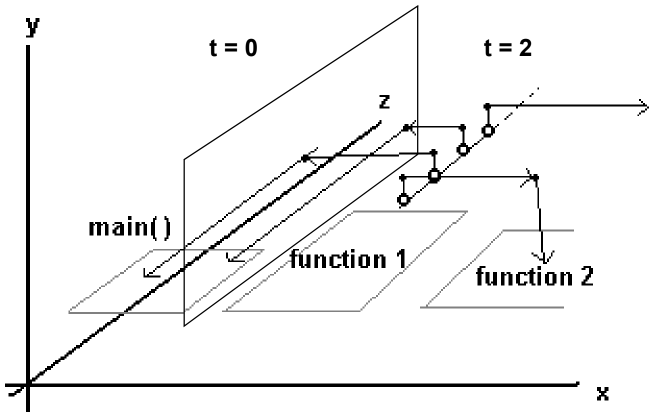

Figure 4: Various returns from function 1.

C's *main* function is stored in the $t = 0$ hyperplane and successive function bodies are laid out along the *x*-axis of the $t = 2$ hyperplane. Hence, the return jumps in the *y*- and *z*-axes cannot accidentally jump into a function body arranged along the x-axis. Instead, control returns via a chain of jumpers to the correct caller. This illustrates an important principle in the design of perspex programs and compilers. The different axes of perspex space can be used locally to support different kinds of operation, giving rise to a local, spatiotemporal modularity in the operation of the machine which precludes accidental operations. This might lead one to consider whether interpolation[6] within the perspex machine should be arranged in an anisotropic way so that interpolation occurs separately in the different axes of perspex space, thereby preserving the different uses of the axes. The question of whether the interpolation should be local or global along an axis would remain open, though the cellular structure of the universal perspex machine[11] would militate in favour of local interpolation.

6. Arrays

Figure 5: shows the layout of a 1D array, nominally indexed by x. The array is stored in the $t = 1$ hyperplane. The figure shows the entry point of the array at the origin of a local co-ordinate-frame with x-, y-, and z-axes. A column of numbers, C_j, lies on the negative z-axis and runs in the vertical, y, direction. Running vertically from $y = 0$, the numbers are the jump return value, an Input/Output (I/O) selector flag, the I/O datum, and the x index. On the positive z-axis there are five layers of perspexes. The middle layer is the data layer which holds the array's data. The two layers above and below the data layer are mirror images of each other. The upper two layers are for data output, the lower two layers are for data input. The perspexes at the z-extreme are jumpers that return control to just after the kth lexical call of the array access.

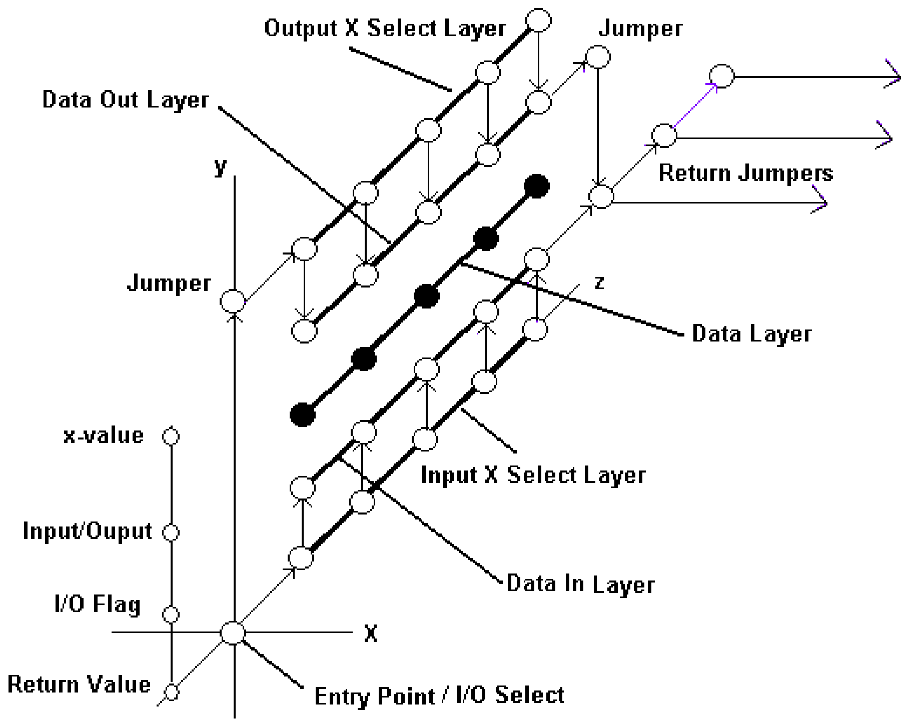

Figure 5: 1D array.

An array access is rather like a function call and return. Firstly, the *I/O Flag* is set to C_1 for input or C_0 for output. If the flag is set for input then the input datum, C_i, is written into the *Input/Output* location, otherwise this location is left unchanged until it is overwritten by the output. The array index, C_x, is written into the *x-value,* and the jump return value, C_k, is written into the *Return Value.* Secondly, control is passed to the *Entry Point* of the array. The *Entry Point* is a perspex that makes the I/O selection. It multiplies the *I/O Flag* by I and then jumps along the z-axis for input or the y-axis for output, where another jumper moves control to the start of the *Input X Select Layer* or the *Output X Select Layer,* as appropriate. The two layers work in the same way, just like a function return. Control passes x units along the selection layer until the *x-value* is decremented to zero. At this point control jumps into the *Data In Layer* or the *Data Out Layer*. The *Data In Layer* takes the *Input/Output* parameter, multiplies it by I, and writes the result into the location immediately above it in the *Data Layer.* The result of the multiplication is, of course, C_i, with element $z_{11} = 1$, which causes a jump in the *z*-axis. The size of the jump is determined by the *Data In Layer* perspex. It passes control directly to the start of the return jumpers which use the *Return Value* to compute a return to the k^{th} lexical call of the array access. The *Output X Select Layer* and the *Data Out Layer* work similarly. Two dimensional arrays are implemented similarly.

There are several important things to note about this arrangement of perspexes. Firstly, the compiler leaves a margin of uninstantiated perspexes around the array. These default to the halting perspex H. Thus, any access beyond the array bounds causes the perspex machine to halt. This behaviour is compatible with C, but is more secure than most C compilers. It is impossible, for example, to execute buffer-overflow viruses for a system compiled using such a perspex C compiler. Secondly, Figure 5: looks rather like a wiring diagram for an array. This is no accident. Physical devices cannot implement a specific computation in arbitrary ways. The geometry of the universe constrains the possible implementations of a computation. The geometry of the perspex machine also constrains the implementation of computations. There is sufficient similarity between the geometry of wire tracks and the motion of electric charge, and the position and motion of perspexes, for similar solutions to appeal to the mind of a human designer. In this way the perspex machine links together the physical and computational aspects of computer design. Therefore, we should expect Computer Aided Design (CAD) tools to be useful in the design of perspex programs.[7] Thirdly, the *Data In Layer* and *Data Out Layer* are constrained to pass control, from whatever position they are at, to the beginning of the return jumpers. This enforces an analogue structure on the jumps so that, self evidently, the length of the jump is proportional to the distance from the destination. Thus, analogue structure emerges naturally in the implementation of a perspex computation. This is important, because the prospects for useful interpolation properties[6] is enhanced wherever perspexes have an analogue interpretation.

7. Arithmetic

Basic C arithmetic is implemented as a product of homogeneous transformations as shown in Table 1. The first column, headed "Operation," shows the kind of arithmetic operation that is being implemented. The second column, headed "Types," shows the types of the arguments to the arithmetic operation. Here, $\mathrm{Var}(a)$ means that a is a variable and $\mathrm{Num}(a)$ means that a is a number. Note that every operation returns a variable. This simplifies the compiler by making it possible to hold intermediate values in variables, rather than having to maintain trees of literal numbers covering every execution path. Unfortunately, this slows down execution of the compiled code where a number is the natural result, say in the more natural $\mathrm{Num}(x) + \mathrm{Num}(y) \to \mathrm{Num}(z)$. However, the priority in this, the first C to perspex, compiler is to achieve compilation regardless of the efficiency of the compiled code. The third column, headed "Implementation" shows a product of homogeneous transformations being assigned to a location, $\vec{z}$, in space. This implements the arithmetic operation. These transformations are defined earlier in this paper. They are all premultyplying, active transformations of co-ordinates. This means that they are, conventionally, evaluated from left to right. If a right to left evaluation is wanted this can be had, as usual, by writing the transpose of the matrices in the opposite order. The final column, headed "Condition," shows a boundary condition, if there is one. In the case of multiplication, failure to observe the boundary condition does correctly compute a zero product, but too much of the matrix is zeroed and this destroys the type information within it. In the case of division by zero the product computes a result, but not one that is compatible with C arithmetic. In these cases it would be possible to trap the breaking of the boundary condition and jump to either a different sequence of perspexes that computes the desired result, or else to a sequence of perspexes that sets an error flag and halts. Thus, the perspex machine, which has no exception states, could emulate the exact behaviour of C, including its arithmetical, or logical, exceptions.

Operation	Types	Implementation	Condition
Assignment	$\mathrm{Var}(x) \to \mathrm{Var}(z)$	$V_x I \to \vec{z}$	
	$\mathrm{Num}(x) \to \mathrm{Var}(z)$	$C_x G \to \vec{z}$	
Addition	$\mathrm{Var}(x) + \mathrm{Num}(y) \to \mathrm{Var}(z)$	$C_y V_x \to \vec{z}$	
	$\mathrm{Num}(x) + \mathrm{Var}(y) \to \mathrm{Var}(z)$	$C_x V_y \to \vec{z}$	
	$\mathrm{Var}(x) + \mathrm{Var}(y) \to \mathrm{Var}(z)$	$V_x G V_y \to \vec{z}$	
	$\mathrm{Num}(x) + \mathrm{Num}(y) \to \mathrm{Var}(z)$	$C_x C_y G \to \vec{z}$	
Subtraction	$\mathrm{Var}(x) - \mathrm{Num}(y) \to \mathrm{Var}(z)$	$V_x S C_y G S_a \to \vec{z}$	
	$\mathrm{Num}(x) - \mathrm{Var}(y) \to \mathrm{Var}(z)$	$C_x G S V_y S_a \to \vec{z}$	
	$\mathrm{Var}(x) - \mathrm{Var}(y) \to \mathrm{Var}(z)$	$V_x S V_y S_a \to \vec{z}$	
	$\mathrm{Num}(x) - \mathrm{Num}(y) \to \mathrm{Var}(z)$	$C_x G S C_y G S_a \to \vec{z}$	
Multiplication	$\mathrm{Var}(x) \times \mathrm{Num}(y) \to \mathrm{Var}(z)$	$M_y V_x M_y G G \to \vec{z}$	$y \neq 0$
	$\mathrm{Num}(x) \times \mathrm{Var}(y) \to \mathrm{Var}(z)$	$M_x V_y M_x G G \to \vec{z}$	$x \neq 0$
	$\mathrm{Num}(x) \times \mathrm{Num}(y) \to \mathrm{Var}(z)$	$M_x C_y G M_x G G \to \vec{z}$	$x \neq 0$
Division	$\mathrm{Var}(x) / \mathrm{Num}(y) \to \mathrm{Var}(z)$	$G M_y G V_x G M_y G \to \vec{z}$	$y \neq 0$
	$\mathrm{Num}(x) / \mathrm{Num}(y) \to \mathrm{Var}(z)$	$G M_y G C_x G G M_y G \to \vec{z}$	$y \neq 0$

Table 1 Perspex implementation of arithmetic with perspex products evaluated from left to right.

No product of perspexes has been found that preserves variables and performs the multiplication of two variables, or performs the division of a number or a variable by a variable. These cases are handled by iterative C functions, shown in Table 2. These function compute the result to a precision fixed by the number of iterations, here five. The functions are known to compile into the basic arithmetical operations that are supported by products of matrices. Thus, the compiler has reached the stage at which it is beginning to bootstrap its own development.

In contrast to the perspex machine being used here, which has a linear instruction, the universal perspex machine[11] has a general linear instruction. This allows it to access arbitrary elements of a matrix so it can construct natural matrix products for all arithmetical operations.

It is interesting to note that the normalisation step in the perspex instruction[4,5] can cause the product of homogeneous transformations to be non-associative. This occurs only where a division by a number other than unity is involved. For example, the fragment, GM_yGG, of the division implementation, computes a reciprocal of y, but $GM_y(GG)$ does not. We have, writing the matrices without their middle two, identity, rows and columns:

$$GM_yGG = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} y & 0 \\ 0 & 1 \end{bmatrix} GG = \begin{bmatrix} 0 & 1 \\ y & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} G = \begin{bmatrix} 1 & 0 \\ 0 & y \end{bmatrix} G = \begin{bmatrix} 1/y & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1/y \\ 1 & 0 \end{bmatrix}. \qquad \text{(Eqn 11)}$$

But:

$$GM_y(GG) = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}y & 0\\0 & 1\end{bmatrix}\left(\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\right) = \begin{bmatrix}0 & 1\\y & 0\end{bmatrix}\left(\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right) = \begin{bmatrix}0 & 1\\y & 0\end{bmatrix}. \qquad \text{(Eqn 12)}$$

Projective geometry is associative. Indeed $GM_yGG \equiv \lambda(GM_y(GG))$ with $\lambda \neq 0$, as required by associativity. But the affine fixing of projective geometry obtained by the perspex normalisation step is non-associative. This non-associative model is used in computer graphics, yet graphics pipelines are assumed to be associative.[9] A fuller analysis of associativity might justify the practice of applying normalisation only once at the end of a pipeline. Alternatively, it might open up new, non-associative algorithms, perhaps for calculating radiosity form-factors via the hemisphere method and its variants.[8]

Multiplication

```
double mult(double a,b) {
  double iter, crment, total, modd, m;

  iter = 5;
  crment = a;
  total = 0;

  if (b<0){
  modd = -1;
  }
  else {
  modd = 1;
  }

  while (iter>0){
    total = total + crment;
    b = b - modd;
    m = 0;
    if (modd < 0){
      if (b>modd)
        m = 1;
    }
    else {
    if (b<modd)
      m = 1;
    }

  if (b = 0) {
    iter = 0;
  }
  else if (m = 1) {
          modd = modd * 1_/10;
          crment = crment * 1_/10;
          iter = iter - 1;
       }
  }
return total;
}
```

Division

```
double div(double a,b) {
  double iter, crment, total, modd, sign;

  iter = 5;
  crment = b;
  total = 0;
  sign = 1;
  modd = 1;

  if (a<0) {
    a = a * -1;
    sign = sign * -1;
  }

  if (b<0) {
    b = b * -1;
    sign = sign * -1;
  }

  while (iter>0) {
    if (a = 0) {
      iter = 0;
    }
    else if (a>crment) {
         total = total + modd;
         a = a - crment;
         }
    else {
      modd = modd * 1_/10;
      crment = crment * 1_/10;
      iter = iter - 1;
    }
  }

  if (sign = -1) {
    total = total * -1;
  }

  return total;
}
```

Table 2 C code to perform multiplication and division – bootstrapped into the compiler.

8. Conclusion

The compiler compiles part of the C language to perspexes. It can handle arithmetic, *while* and *for* loops, *if-then-else* conditionals, one and two dimensional arrays of numbers, and non-recursive function call and return. In principle the compiler can be extended to handle the whole of the C language or, indeed, the whole of any Turing computable language. C was chosen as the source language for this first, perspex, compiler because there are cross-compiler pathways to C from many computer languages. Thus, one can envisage compiling many, if not all, existing scientifically and commercially significant programs into perspexes.

The development of the compiler has exposed some interesting design principles. Firstly, if an array, or other data structure, is stored in an otherwise empty hyperplane then accessing it outside of its bounds involves a jump into an uninstantiated location of the hyperplane and this halts the machine, because every uninstantiated point in space contains the halting instruction by default. Therefore, it is impossible to execute buffer-overflow viruses in a program complied on such a perspex compiler. More generally, this highlights the importance of the geometrical arrangement of data structures.

Secondly, the compiler generates matrices that carry type information within them. Thus, in at least assignment, addition, and subtraction, an object carries its type information at all times. Furthermore, a cast from one type to another is simply a matrix transformation. We expect that this will make it easier to prove the correctness and completeness of type polymorphisms. It also raises an interesting, theoretical, possibility. Perspex operations, except the perspex halt, are continuous.[6] Therefore one can construct matrices that lie mid-way between types. One could, for example, construct a spectrum of objects, by linear blending, that passes from a variable at one end of the spectrum to a number at the other. This might be useful in constraint satisfaction systems where a variable becomes constrained to a constant number during the satisfaction process.

Thirdly, the perspex machine has no exception states, but it can emulate exceptions in a standard, computer language by setting a flag and halting. However, if it is allowed to operate without emulating exceptions then there are no bottle necks on setting status words or error flags. The machine can be made massively parallel. As the perspex machine has a single instruction it would seem to be particularly well suited to parallel implementation, though one might prefer to implement the machine in terms of the general-linear, perspex instruction,[11] rather than the linear instruction used here. This is because the general-linear instruction can access arbitrary elements of a perspex, leading to natural implementations of all standard, computer operations. The perspex machine is a Single Instruction, Zero Exception (SIZE) machine.

It is interesting that arbitrary, computer operations can be laid out geometrically[4] and that this can be done by a compiler. As all of the operations in the current perspex machine are perspective transformations and jumps, this raises the possibility of implementing programs on an optical computer. By contrast, the universal perspex machine is better suited to approximate simulation on a digital computer.

It is, perhaps, surprising that the product of homogeneous transformations, involving normalisation, can be non-associative. As computer graphics uses such a model of projective geometry, it is open to question how much of a graphics pipeline can be optimised by pre-compiling it. Non-associativity might prevent some optimisations. However, as all contemporary pipelines apply normalisation only once, at the end of the pipeline, this has no impact on the contemporary practice in computer graphics. One might hope, however, that an analysis of non-associativity would either give a formal basis for the current practice, or else open up new graphical methods where perspective projections are carried out several times during a pipeline. For example, the computation, or approximation, of form factors[8] in radiosity might benefit from this approach.

In conclusion, we have demonstrated the compilation of C programs into perspexes and have identified new, geometrical principles of language design, as well as identifying an area of potential research on non-associative graphics pipelines. We finish by observing that the perspex machine is the ultimate Reduced Instruction Set Computer (RISC), it is a Single Instruction, Zero Exception (SIZE) machine. This might make it particularly well suited to implementation as a parallel machine.

References

1 J.A.D.W. Anderson, ed, *POP-11 Comes of Age: the advancement of an AI programming language* Ellis-Horwood (1989).

2 J.A.D.W. Anderson, "Representing Geometrical Knowledge" *Phil. Trans. R. Soc. Lond. B.* 352, 1129 - 1140 (1997).

3 J.A.D.W. Anderson, "Exact Numerical Computation of the Rational General Linear Transformations" in *Vision Geometry XI* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 4794, 22-28 (2002).

4 J.A.D.W. Anderson, "Perspex Machine" in *Vision Geometry XI* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 4794, 10-21 (2002).

5 J.A.D.W. Anderson, "Perspex Machine II: Visualisation" in *Vision Geometry XIII* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 5675, 100-111 (2005).

6 J.A.D.W. Anderson, "Perspex Machine III: Continuity Over the Turing Operations" in *Vision Geometry XIII* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 5675, 112-123 (2005).

7 James Anderson, Matthew Spanner & Christopher Kershaw "Perspex Machine IV: Spatial Properties of Computation" in *AISB Quarterly* Issue 121, Summer (2005).

8 M.F. Cohen & J.R. Wallace, *Radiosity and Realistic Image Synthesis,* Academic Press (1993).

9 J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, *Computer Graphics Principles and Practice,* 2nd edition, Addison-Wesley (1990).

10 C.J.A. Kershaw & J.A.D.W. Anderson, "Perspex Machine VI: A Graphical User Interface to the Perspex Machine" in *Vision Geometry XIV* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 6066 (2006).

11 J.A.D.W. Anderson, "Perspex Machine VII: The Universal Perspex Machine" in *Vision Geometry XIV* Longin Jan Latecki, David M. Mount, Angela Y. Wu, Editors, Proceedings of the SPIE Vol. 6066 (2006).