GSoC Proposal- Implementing smarter boolean operations on vector shapes by using an efficient algorithm for Bézier curve intersection

Project for Krita, under KDE organisation

Tanmay Chavan
Pune, India (+5:30 UTC)

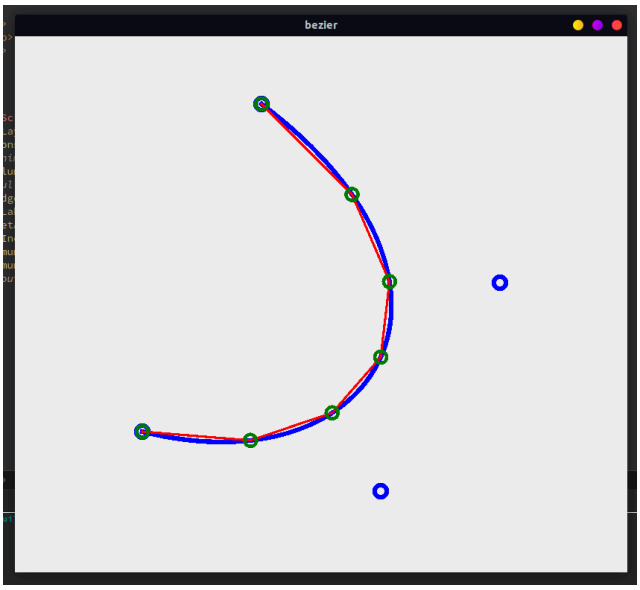
chavantanmay1402@gmail.com
Earendil_14 on Freenode IRC

Abstract:

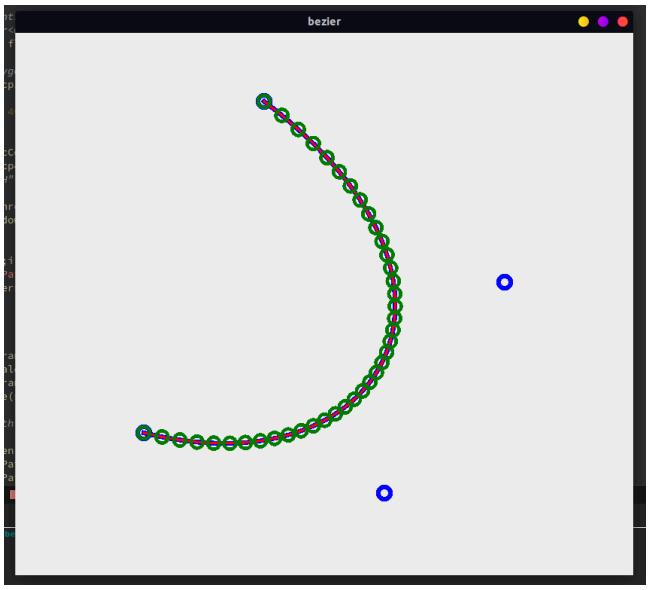
Krita has functionalities where we can perform several boolean operations like unite, intersect, subtract on two or more vector shapes. These shapes can contain vertices, segments, and curves. In order to perform boolean operations on shapes, we need to find their point of intersection. Now, Qt uses a special kind of curve known as Bézier curve, which is rather easy to plot. However, finding the intersection points of these curves can get rather difficult. The default algorithm in Qt does it by an approximation, where it constructs an (open ended) polygon which provides a pretty good approximation. But this leads to an excess number of nodes. I propose to implement another algorithm which involves finding the roots of the polynomial representing the curves, which will give us an exact solution.

Goals:

- To reduce the number of nodes generated while performing boolean operations on vector shapes
- To implement the implicitization algorithm instead of the pre-existing algorithm
- To modify the way Qt handles intersections for actually computing urve intersections



With less nodes
The polygon formed does not represent a curve
Properly, but has few nodes.



with more nodes
The polygon formed is a very good
Approximation of the curve, but has
Many nodes.

Above images are screenshots from a toy project i created,
https://invent.kde.org/earendil/bezier-flattening

Relevant bugs/wishes:

https://bugs.kde.org/show_bug.cgi?id=400521

Concept:

Qt uses Bézier curves to plot all kinds of curves; including elliptical and hyperbolic curves. Bézier curves can be defined by a parametric equation of binomial polynomials ,where the coefficients of the terms are the coordinates of the control points. Control points define the trajectory of the curve, and are the reason why these curves are used so extensively

in Computer Graphics. The extra nodes are generated because of the default algorithm for finding Bézier curve intersections. It generates near adjacent points on the curve decided via a threshold. In this case, we could approximate the curve to be a straight line joined by the start and end points, and thus reduce the Bézier curve to a simple open (or closed) polygon. However, as this method converts it to a polygon, it can be proven that for a finite number of splits on the curve, the intersection point generated will not lie on the actual curves, unless it is on one of the vertices of the polygon and there is no point of inflection in between. (consider two points: the straight line joining them is the shortest possible path between them. As we are considering a curve, which will not be a straight line and thus not be the same as the line. Hence, we can assume that a point lying on the line segment which is not one of the end points will never coincide with a point on the curve.) We could actually try to move the curve to pass through the intersection point. But that would require an efficient offsetting algorithm, which is also a challenge.

There is another method to find the intersections of two Bézier curves, known as *implicitization*, which is briefly explained as follows: We convert the parametric form $x = f(t),\ y = g(t)$ of the equation representing the curve to an implicit form $f(x,\ y) = 0$. To do this, we basically find the common roots of the two parametric equations by using the properties of *resultant*, and then we convert the individual equations from explicit to implicit form ($x = f(t)$ into $f(x, t) = 0$, for both x and y). Then, for two-curve intersection, we have to plug in the values of the parametric equation $x = f(t),\ y = g(t)$ into the implicit form of the other curve $f(x,\ y) = 0$(Note that we only have to implicitize one curve to find the intersection). Up until this point, we were dealing with *exact* values. Now, the resultant polynomial is going to be of 9th degree, for

which there is no method to find the exact roots. Hence, we will use newton's method to calculate the values of t at which they intersect.
(The necessary proofs and equations along with an example are provided here, in section 17)

The benefit of using the implicit form is that we can easily find the roots of the equation using linear algebra. This method is almost twice as fast as the Bézier clipping method (the predominant algorithm to find Bézier curve intersection), and ten times faster than the Bézier subdivision method. However, this algorithm loses its edge when the degree of Bézier curves cross 3, and obtaining an exact solution becomes far too strenuous, if not impossible, and we have to use approximations. However, Qt implements its curves only as cubic Bézier curves (not even quadratic, it elevates it to a cubic curve), for which our method is efficient as well as accurate.

Another important point is that if we don't use Gaussian elimination for common root finding, the Big-O complexity of the algorithm remains constant. That is, it won't increase with the increase in the value of the coordinates of control points (obviously, this only remains true if we limit ourselves to cubic curves, which is indeed the case here). So the main factor regarding complexity will be introduced by the numerical methods used. This can be optimised by using different strategies according to the number of intersection points available.

Implementation Plan:

In Krita, the shapes are stored as KoShape objects, which contain all of the geometric data using the QPainterPath class, which has 3 elements: moveTo, lineTo, and curveTo. Now, the QPainterPath

object is further broken down into QWingedEdge object, which essentially contains all of the data as ordered lists of vertices and segments. This is the step where a curve is converted to an open polygon (in the addPath member function, to be exact). After this step, it proceeds to perform a procedure to find line-line intersections. In my method, the Bézier curve will remain a curve, and I will add another algorithm to the function to find intersections to find curve-curve as well as line-curve intersections.

I. Main algorithm implementation:

For generating the resultant matrix, some functions need to be implemented. First, it would be good to know the number of roots, or if the roots exist (should be rejected as we know the curve we are plotting exists, and that every parametric curve can be represented in it's implicit form). For this, we need to create a 3 x 3matrix with its elements being generated by the resultant formula through auxiliary equations, provided by the Eigen library which is already used in Krita. As we are going to be dealing with only Bézier curves (and of the third degree), we don't have to create an algorithm for the general case; one function to calculate the expression in each element of the matrix, and another to find the value of it's determinant will suffice. (to simplify the process, we could use Gaussian elimination to get the rank of the matrix, but it would require some extra code and would introduce another non-constant factor in the Big-O notation). After that, we need to change the explicit component of $x = f(t)$ to $f \circ (x, t) = 0$ (this can be easily done by subtracting x from the expression containing t in the elements of the previously calculated determinant). After we generate the implicit equations and plug them in the parametric

equation of the other curve, we get a polynomial with a pretty high degree (maximum 9, minimum 4). Finding the exact roots for a degree 4 polynomial is strenuous but still possible. However, after degree 5, we cannot find the exact roots via our classic algebraic methods. To tackle this, we need to use numerical methods, which provide a reasonably good estimation of the actual roots. However, as we're dealing with Bézier curves, the variable parameter t will vary from 0 to 1. Now, for most of the cases the number of intersections will be 1 or 2, and rarely 9. Hence, we could use Sturm's theorem to calculate the number of roots in the period . For Strurm's theorem, we will have to calculate the derivatives of the initial polynomial until we reach a constant value, then plug in the values of the limits (0 and 1). The number of roots can be evaluated by checking the number of times the sign of the If there is only one root, we could use Newton's method. If there are two or three roots (relatively evenly spaced, might be apparent in Sturm's chain), we could use exclusion and eliminate intervals with no roots. Finally, if there are too many roots, like a 9 degree polynomial, we could use Aberth's method which is readily implemented in the C library MPSolver, However, rather than adding a new dependency, we could as well use GSL, which also has functions to find all the roots Both the libraries use different algorithms to tackle the problem. The QR reduction in GSL is of time complexity$O(n^3)$and space complexity $O(n^2)$, while MPSolver has$O(n^2)$ and$O(n)$ respectively . However, for smaller orders, one can expect better performance from GSL due to optimisations. Some research would be required to determine whether we can rely on the GSL implementation, and decide accordingly. Knowing the number of roots beforehand and given they are one or two would significantly reduce the computation

needed, and if they fail to find a solution, we could always use the libraries mentioned above.

As a lot of the above code will have to be written mainly from scratch and by reusing Krita code, a new class must be added to handle the intersections (KoIntersections, maybe?) This class will handle the matrices and subsequent methods to generate implicit equations. Apart from this, it would be useful to include all of the numerical methods in a separate class (KoNumericalMethods), for better modularity. We will have to implement two different unit tests for the individual classes as well.

II. New intersection algorithm implementation

Qt uses the tree data structure to implement intersections, via a multitude of functions and a few classes. However, the base for determining the initial intersections lies by constructing a rectangle around the element and checking for overlaps. We can use the same bounding box method for finding intersections with other elements. The krita class KoRTree has already implemented an R-tree which is suitable to find intersections via an efficient BoundingBox method. The class mentioned above would help implement the Qt code inside Krita and might as well get rid of the need to use most of the Qt intersection code. Hence, in the algorithm, the only main change we'd need to introduce is to curve-line and curve-curve intersections. For this purpose, we could add a new ID element (similar to ID in QPainterPath). Hence, we could add another class containing (but not limited to) curveCurveIntersections() and

curveLineIntersections(), and merge them easily with the intersection algorithms in QPathClipper.

As mentioned in the algorithm, only one curve needs to be implicitized. We could implement the implicitization on a "need-to-know basis", where we will implicitize the curve only if it's bounding box has intersected with another element's bounding box. Then, we could simply add the implicit equation to a register, so we could use it directly for future purposes.

Also, if we were to import Qt code within Krita, I'd have to implement another set of unit tests where it verifies if the intersection points are generated correctly and all the corner cases are covered. Apart from this, some checks are necessary to verify if the numerical methods are capable of doing the job in some situations; for example, if Newton's method fails and keeps converging to a root found previously, we could set a limit on time/iterations to notify whether we should move on to Aberth's method. As the exact nature of these cases cannot be determined with full confidence before implementing the code, some unit tests would provide a safety-net to switch to a different approach. Apart from this, as Qt doesn't truly support boolean operations on curves, I'd have to set up some more unit tests to verify if the boolean operations work as expected.

III. Qt Modifications:

Most of the code to be modified lies in two classes: qpathclipper and qpainterpath, and it might as well be possible to achieve our goal by modifying only two files (thus making it easier to

add the patch to Qt itself). For further speeding up the process, we could use the bounding box method to verify if the two boxes intersect; if not, we can move on (the bounding box will always be larger than or equal to the convex hull of the curve, which in turn always contains the entire Bézier curve). Also, a lot of code regarding the above method is already implemented in Krita KisBezierUtils, which could be reused.

Regarding the unit tests, as most of the code for intersections lies behind the Qt API, new unit tests will have to be written. However,as we aim to replace the features *behind* the Qt API, it is highly probable that the external unit tests may perform in the same manner as before, with the notable exception of extra nodes being generated and their associated actions. So, I'll mainly need to write new unit tests for the internal code, which although might be needed to be written roughly from scratch, should be fairly contained, and would work without much further refactoring. Amongst these, I'll have to add unit tests for tst_qpainterpath, where I'll have to add tests for curve-curve and line-curve intersections. More importantly, tst_qpathclipper tests are crucial to verify if the algorithm truly works, as it contains several tests for Bézier curve implementation directly as well as indirectly (to plot ellipses, and composite shapes) . It also contains functions to verify if filling algorithms work correctly.

For the above approach, I was planning to copy the two files (qpainterpath.cpp and qpathclipper.cpp) in the Krita codebase, keep the relevant code and remove the ones not needed for our current task and let them use the original Qt modules. However, the problem with this approach is that the code relies on further Qt private

modules, which are subject to change at their discretion; thus it is not really reliable, and could potentially add further load for maintenance.

Another approach would be to eliminate the need for using qpainterpath itself. This is possible, as KoShapes contain almost all of the necessary data required to perform boolean operations. For this, I'd have to set up new code to find the intersections, and set up a new method to get the shapes converted to another form which would be efficient for finding intersections and other boolean operations (For this purpose, trees seem to be used; it can be created by reusing KoRTree as well as by modifying Qt's own QWingedEdge). The former step is easy, however the latter could potentially be a problem. Krita's method of storing data of the shapes is different from Qt. However, as all the necessary information is contained in KoShapes, it might be a better solution, although it would require more time. However, copying Qt code and managing the dependencies would work perfectly for our purposes, and we need not create another intersection algorithm. Hence, we'll proceed with the former method.

IV. Testing:

I am going to be writing documentation and creating unit tests simultaneously while writing the code, so most of the problems will be seen and solved in the initial stages. However, this phase will include testing the modifications rigorously from the perspective of the user. It is important to verify if the added code plays well with the Krita codebase and does not generate any errors or conflicts. I'll

also check if all of the documentation makes sense and if any further unit tests are to be added (for verifying the new algorithm works properly with other features).

Rough Timeline:

(As the working time period for GSoC has been reduced to half, I'm planning to start two weeks earlier)

24 -30 May	Interact with the community and the dev team. Discuss and create a proper strategy to tackle the problem with my mentor. Start writing about my progress and the status of the project on my blog linked to PlanetKDE.
31 May - 6 June	Start implementing the implicitization and root finding algorithm: Resultant matrices and generating the implicit functions, and write documentation for the same.
7 - 13 June	Start implementing the implicitization and root finding algorithm: setting up numerical methods for finding the roots, and adding unit tests for implicitization and root-finding. Finish documentation for root-finding.
14 - 20 June	Start implementing the new/modified intersection finding algorithm: Create a proper plan to merge pre-existing line-line intersection code with new code. Starting to write the documentation regarding new intersection functions.

21 - 27 June	Start implementing the new/modified intersection finding algorithm: Implement the modified manner in which the intersections will be calculated, especially curve-curve ones.
28 June - 4 July	Start implementing the new/modified intersection finding algorithm: Add a bounding box method to reduce calculations. Add relevant unit tests. finish documentation for bounding box as well as the entire new approach.
5 - 11 July	First evaluation. Start integrating the implemented functions in the main codebase: Start planning on how to integrate the intersection algorithm with the codebase, and to select the proper approach for the same.
12 - 18 July	Start integrating the implemented functions in the main codebase: Implement a stable method to manage private libraries used by the Qt intersection functions. Write unit tests to test if the modifications in the Qt code would work as expected with Krita. Write documentation on how to work with the new approach.
19 - 25 July	Check if all the documentation written is complete and is easy to understand. If not, improve it. Try running all the unit tests and check whether the code works as expected and if not, add some more unit

	tests accordingly.
26 July - 1 August	Test the code rigorously via using the feature along with other functions like brushes and layer mechanisms.
2 - 8 August	Test the code. Extra time, if need occurs
9 - 16 August	Test the code. Extra time, if need occurs

Personal Details:

Hi! I'm Tanmay Chavan (earendli_14 on the IRC), from Pune,India. I'm currently in my sophomore year pursuing a Bachelor's degree in Computer Engineering, at Pune Institute of Computer Technology (Affiliated to Savitribai Phule Pune University). I have a profound interest in coding, and love the concept of open source programs for the benefit of the community.

I recently took a course in Computer Graphics, where we had to learn and implement several basic Computer Graphics programs like midpoint circle drawing algorithm, various clipping algorithms, optimised filling methods, and transformation of 3D shapes using homogenous matrix transformations. More importantly, we also studied interpolations

and approximation methods regarding B-spline and Bézier curves (a copy of the syllabus can be found here). As a plus, we were asked to implement all of the above codes in Qt, to make us learn the platform and produce modular code. Because of this, I chose this project, as I would be able to apply my skills learned at university. I also have extensively examined the way Qt handled intersections, and spent a lot of time exploring the perfect algorithm for our purposes.

I have not applied to any other organisation than Krita, nor do I plan to send a proposal to a different organisation.

As for the time conflict, our end semester exams will be conducted in the first two weeks of may, and will be free after that. So there shouldn't be any conflict during the time period.

Prior Contributions:

https://invent.kde.org/graphics/krita/-/merge_requests/757

I plan on being active with the Krita community after GSoC as well, as it is a really good learning experience : rarely does a student get a chance to be involved with a software with such a huge codebase. I'd also give my full effort for the project, and regularly keep in touch with the mentor and push MRs as well for smooth evaluation.I will also add my blog to PlanetKDE and update it regularly. I'd really like to work on the project. I believe I have understood the topic quite well and would be able to implement the required functionality in the given time frame.

References and links:

1. Sederberg, Thomas W., "Computer Aided Geometric Design", (2012) https://scholarsarchive.byu.edu/facpub/1/

2. MPSolve, https://numpi.dm.unipi.it/software/mpsolve

   Bini, Dario A., Fiorentino, Giuseppe, *Design, analysis, and implementation of a multiprecision polynomial rootfinder*. Numerical Algorithms 23.2-3 (2000): 127-173.

   Bini, Dario A., and Robol, Leonardo. *Solving secular and polynomial equations: A multiprecision algorithm*. Journal of Computational and Applied Mathematics 272 (2014): 276-292.

3. Pomax, A Primer on Bezier Curves, https://pomax.github.io/bezierinfo/#reordering