OptiMixer2_14

D.Dekkers (cThrough), 08102002

OptiMixer2_14 1
Abstract 1
Context 1
OptiMixer from the designers perspective 4
Background on optimisation 5
Solution envelope 5
Evaluation criteria 6
Variation mechanism 7
Optimisation algorithm 8
Conclusion 9
FAQ (Added 06062003) 10
What are OptiMixer/FunctionMixer/RegionMaker/...? 10
What does OptiMixer do? 10
Is there a commercial or a trial version available? 10
How does the collaboration MVRDV/cThrough work? 11
How should i interpret the flickering of the coloured pixels? 11
Is this a neural network? 11
What were inspirations for OptiMixer? 11
Why do you use coloured blocks? 11

Abstract

MVRDV performs research on formalised methods for urban planning and design. To contribute to these goals, cThrough developed a software application named OptiMixer. OptiMixer is a design tool, it consists of a formalised process that evaluates, variates and optimises spatial envelopes. The specific implementations of spatial envelopes are controlled by the designer, in the role of the user of the software. This control is not expressed directly, but indirectly, via a set of parameters. The number of used parameters, the precise definitions and the relative degree of influence of each parameter can be set by the user. Using the parameters, OptiMixer can evaluate spatial configurations. Furthermore, it can create variants. Combining the two, it is capable of producing optimisations.

Context

To define the context of this project, we looked at various examples of computer use in urban design and planning. A number of examples are:

- Internet
  The quantitative superlative of information providence. A large amount of complex, incoherent, in quality variating information, as well as a an abundance of methods to search and visualise this information.
- Modelling, simulation, visualisation and animation software
  A given design is shown in an aesthetic or clarifying way. This category of applications has the tendency of encapsulating more and more knowledge from (physical) reality via the mathematic models used. In the early days of computer graphics, it was considered an achievement when a pixel with a given colour was visible at the correct position on a television monitor. Shortly after, the rules of perspective were added, followed by reasonably correct lighting models. Nowadays, a multitude of phenomena like dynamics or acoustics are modelled realistically.
- GIS (geografic information systems)
  Large amounts of geographically referenced information (data identified according to their locations) is stored. This information can be manipulated, evaluated and viewed in various ways. A GPS system as used in car navigation systems is a relatively simple example of a GIS.

- Computergames
  Real-time strategy and simulation games are made with the intention to amuse but are relevant for our work since they present dynamic, interactive models of reality.

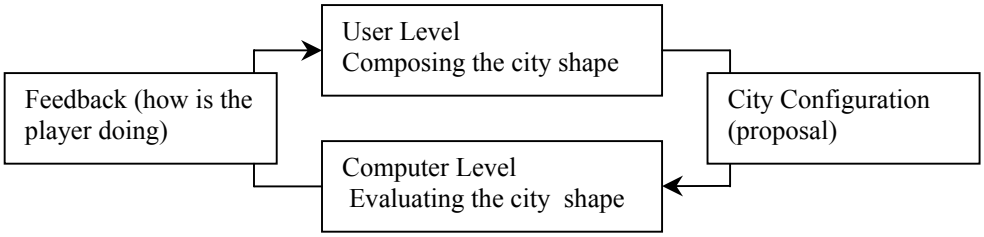
Examining these applications, a number of observations can be made:

Firstly, there is no such thing as a single program *for everything.* Points of interest vary, according to the nature of the design problem. For a sonic wall or a music hall, acoustic simulations can be used. For the placement of oilplatforms, a GIS may be used. For OptiMixer, we accept this issue and create a modular system. As designaccents change, different modules are worked out in finer detail. Furthermore, information is added on a need-to-know basis. A direct connection to the Internet is only useful if the consequences of various types of information on the design is formalised.

Secondly, the applications do not help to actually *design* in an early stage. A design must be elaborated to some degree before a computer can contribute anything substantial. In OptiMixer, this issue is dealt with by formalising the rules that make up the design from the beginning. Design and design forming rules are created simultaneously. Observe that there is no principal differentiation between statistical, aesthetic or functional rules.

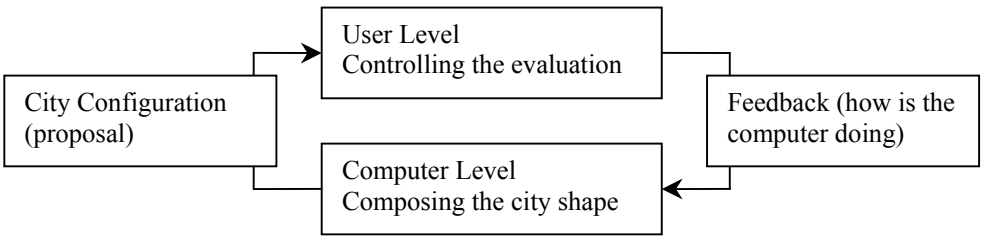
Thirdly, with applications that include some sort of evaluation mechanism, exactly this evaluation remains hidden under the surface. In OptiMixer, the defining process of the various evaluation principles is the most important interface between the designer and the software. This observation touches the heart of OptiMixer, and can be illustrated by a comparison between SimCity and OptiMixer:

After playing SimCity for a while, a player often develops some curiosity concerning the way the game evaluates the proposed city configurations. Sometimes this evaluation is fairly simple, a high crime rate exists, so more police stations should be built. Sometimes the evaluation is quite opaque, like the way infrastructure is evaluated. The following diagram illustrates the approach SimCity takes:



The player of the game places units from various unittypes (roads, residential buildings, commercial buildings, rails, pipes, etc, etc). While he/she is doing that, the current city configuration is sent to the computer program that evaluates the city and sends feedback back to the game player. This evaluation is fixed, and is implemented by the development team at Maxis, the creators of the game. The feedback can consist of indexnumbers (‘overall crime rate’), maps (‘pollution map’), alertboxes (‘water shortage detected’), etc. The player sees and interpreters this feedback and acts accordingly. Units are removed, added or repositioned. The configuration is sent to the program, an evaluation is done, and feedback is again given. Hopefully for the player, the overall city ‘score’ has improved.

With OptiMixer it is, in a way, just the other way around:



A designer using OptiMixer, focuses on the details of the evaluation. This amounts to formalising and adding demands that must be met by the configuration. Because different demands have a different level of significance, and can even be contradictory, a weight factor is added to every evaluation. The progam is continuously searching for configurations that result in high scores and consequently takes over the role of the player in SimCity. The computer is not a very smart ‘player’ (it just tries all kinds of different variations), but it is very fast.

OptiMixer from the designers perspective

Typical usage of OptiMixer amounts to a number of steps.

1. Define your category. It is helpful to start with a word or a term. For example ‘risk’, ‘efficient infrastructure’, ‘diversion’, ‘cosiness’ or ‘leisure fulfilment’.

2. Define the unittypes you need. The unittypes are the building blocks of the city. You can choose to define the set of unittypes by gathering data, looking at the real world. But you can also be inspired while analysing your evaluation (step 3 below). While you were thinking about ‘risk’, you think of a nuclear power plant. So nuclear power plants should be added as a unittype. If you are working on (say) a huge tower filled with live stock, you might need pigs:

   [picture berlage pig, to be inserted]

   In this stage, you also think of the scale and the resolution of your design.

3. Define your evaluation. This is by far the hardest part. It involves a thorough analysis of your chosen category. What is ‘risk’? What elements influence it? A city with a nuclear plant is riskier than a city without one. But a city with a nuclear plant close to dense residential areas is even more riskier. After you analysed your category, try to formalise it, make it countable. The computer should be able to evaluate every single city (randomly generated) and produce a score for your specific category. City X has a risk-score of 10, city Y has a risk score of 3021. Say you are blessed with the category ‘infrastructure’. Then your task would be to make a formalised, countable evaluation to define whether a town shows an efficient infrastructural system. That is an unrealistically complex task. So.. try to find the essentials. From the thousands of different elements that influence infrastructure, try to find those three or four that are the most important, that have the most effect. That choice is difficult, it is based on intuition, creativity, professional knowledge and plain common sense.

4. Add a slider. The slider principle is the same for every category. The purpose of the slider is to give control to the user of the program. If the slider is dropped to zero, the program will not consider that category <u>at all</u>. So you could have made a parameter ‘risk’, but the controller of the program could choose to completely ignore the risk aspect in his/her city. It also allows you to ignore the parameters of your fellow parameter-definers. The other extreme is to slide the ‘risk’ slider all the way up, and all the other sliders down. The program will <u>only</u> consider the risk factor, and nothing else, and would produce the ultimate no-risk city.

Background on optimisation

The overall structure of OptiMixer (and optimisation techniques in general) can be clearly distinguished into four components:

- Solution envelope
- Evaluation criteria
- Variation mechanism
- Optimisation algorithm

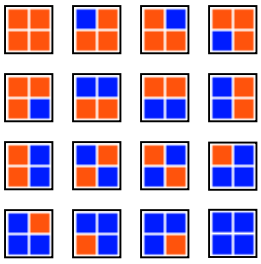
In the remainder of this article these general components will be further elaborated and illustrated by examples from OptiMixer.

Solution envelope

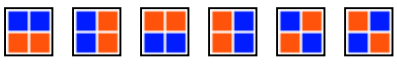
The spatial envelope in which you can work consists of a three dimensional, orthogonal, homogenous grid of cubical shapes, called *voxels*. Every voxel has exactly one function applied to it. The voxelsize together with the number of voxels over the three main axes define the resolution of the world. A function application to a single voxel is called a *unit*. Units are of a specific type, the *unittype.* Examples of different unittypes are housing, offices, parks, industry or infrastructure. A unit is a concrete instance of a *unittype* (a house, a park, etc).

The solution envelope consists of the complete set of all possible valid configurations. In this context, the word 'valid' implies that no qualitative demands are imposed on a configuration, every solution that falls within the degrees of freedom of the model suffices. The solution envelope serves to confine the assignment into a defined space. A lower number of degrees of freedom, or a lower degree of resolution within a degree of freedom will lead to smaller solution envelopes.

A simple example which is inspired by the OptiMixer. In this example only two possible unittypes are found in a 2x2x1 solution envelope. The envelope allows for a limited 16 ($2^{(2x2x1)}$) solutions:



The solution envelope can be reduced to six valid solutions if we state that both unittypes should have exactly two occurrences:



A more expected envelope would consist of 100x100x5 voxels, of which each space can randomly contain 8 unittypes. Therefore there are 8^{50000} valid possible solutions. This is such an inconceivable number, that the solution envelope can only be imagined as a gigantic 'cloud of possibilities' where every point is a complete urban solution in itself. In this cloud, similar solutions can be found close to each other, where the difference may be as small as one unit.

In OptiMixer the solution space is defined by the user. It consists of the size of a voxel, the number of voxels in the direction of the three main axes and the set of unittypes.

Evaluation criteria

The evaluation criteria give a qualitative measure of solutions from the solution envelope. For every solution, the evaluation criteria give a unique score, a grade point, making solutions comparable. Using the evaluation criteria, the designer distinguishes ‘good’ from ‘bad’ solutions. Because OptiMixer is a formalised process, evaluation criteria should be elaborated in detail.

Examples of evaluation criteria are: diversity (to what degree are houses and offices alternated), noise-hindrance (to what degree are different functions spatially organised according to their noise sensitivity) or sunlight (to what degree do functions receive adequate sunlight).

To handle the notion of ‘qualitative measure of solutions’, we chose to use a divide and conquer strategy. The final score is subdivided into subscores (resulting from subevaluation criteria). Every subevaluation criterium is equiped with a weightfactor, resulting in a *parameter*. If S is the solution space, s an element of S, f_1 through f_n are the subevaluation criteria and α_1 through α_n the weightfactors, the total score f of a solution is:

$$f(s) = \alpha_1 f_1(s) + \ldots + \alpha_n f_n(s)$$

A (sub)evaluation criterium is of a specific type, an *evaluationtype.* At any one moment there is always a fixed collection of existing evaluationtypes available to the designer. This set is extended constantly, since there are always design principles that can not be expressed using the current set of evaluation types.

A number of examples of available evaluationtypes are:

- EvaluatorCounter
  An EvaluatorCounter counts for a given unittype the total number of occurrences of units of that specific type. If the current number of occurrences deviates from a given target amount, the score is lower. By adding an EvaluatorCounter for every unittype, the desired quantitative demands can be expressed without adding demands concerning the geometrical partitioning of the units. Furthermore, a demand like ‘minimise asphalt’ can be expressed (set target amount of infrastructure to zero).

- EvaluatorDistanceFixedPoints
  EvaluatorDistanceFixedPoints adds all distances between instances of a given unittype to a list of fixed points. As an example, this evaluationtype makes it possible to minimize distances from houses to infrastructural junctions.

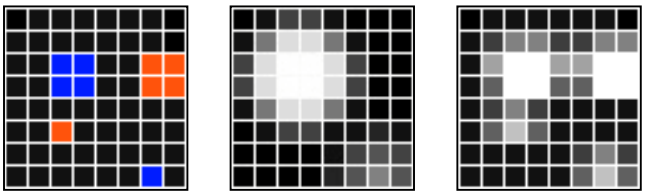
- EvaluatorIntrinsic
  EvaluatorIntrinsic is a purely geometric evaluatortype. Occurrences of a chosen unittype receive a score based on their positioncoordinate. A possible application of this type is a cost penaltyfunction for high-rise buildings. Higher units (with a higher z-coordinate) cost more, units will be placed on ground level.

- EvaluatorInfluenceMap
  Using an EvaluatorInfluenceMap, one can express influences from one unittype to another unittype. Firstly, a list of *generators* is created. A generator represents a source of influence. The local influence of a source can be expressed in a three-dimensional matrix with influencevalues, an *influencemap*. Every occurrence of a unittype that has been defined as a source in a generator, adds this matrix to a global *accumulationmap*. The accumulationmap thus represents the global summation of all these local influences. For different influencetypes, different accumulationmaps exist, think of noise or shadow.

  In the actual EvaluatorInfluenceMap a different (or the same) unittype is chosen which is affected by the influence in the accumulationmap. Think of houses that are affected by the presence of noise or shadow. All these influences are added and form the final score.

  Two examples of influenceevaluators are noise and shadow. Noise shows a logarithmic decline over distance, the centre of the influencmap (the matrix) has the highest value (produced dba level directly in the

source), the corners of the influencemap have the lowest values. The influencemaps for shadow are not symmetrical. Shadow is cast mainly to the north and under the source and to a lesser degree to the east and west. The figures shows the accumulationmap for noise (centre) and shadow (right) and the built areas that produce the influences. Red and blue units generate shadow, only blue units generate noise.



- EvaluatorTransitions
  The EvaluatorTransitions evaluatortype determines the endscore by counting transitions between two definable unittypes.
  This evaluatortype can be used to create mixing or clustering. In the first case, transitions between different unittypes are appreciated (housing/offices, or housing/park), in the second case transitions between identical unittypes are appreciated (housing/housing, park/park).

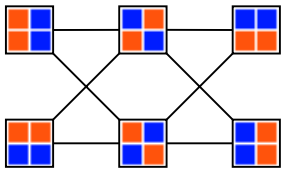
- EvaluatorAdjacencyCount
  The EvaluatorAdjacencyCount type gives control over clustering typologies. A unittype is given, and for every possible number of neighbours of an identical type, an appreciation or disapprovement factor is given. An appreciation of exactly zero neighbours will create single units. An appreciation of exactly two neighbours results in snake-shaped clusters and a positive appreciation of six neighbours gives massive constellations.

Variation mechanism

The variation mechanism offers the possibility of deriving one or more new solutions from an existing one. Small modifications to an existing solution are performed and the new solution is evaluated. Only solutions that produce better scores are accepted. Solutions can thus evolve to solutions with higher scores.

A variationmechanism can produce a large set of possible variations from which a solution is arbitrarily chosen (a fine-grained variationmechanism) or it can follow a predefined deterministic path through the solution space. Both extremes have advantages and disadvantages. A fine-grained variationmechanism can move freely through the solution space and find unexpected solutions (that show a hidden, not expected quality), but it might be relatively slow. A deterministic variationmechanism will find solutions faster, it has built-in intelligence, and the solutions will always show traces of this knowledge.

OptiMixer implements different variationmechanisms. An often used fine-grained variationmechanism is for example the random change of a unit at a random position. A different variationmechanism is the swapping of two arbitrarily chosen units. For this latter case and the above mentioned simple example the following scheme can be made:



Every connection gives a possible path that the variationmechanism can follow through the solution space. Every connection made, offers the variation mechanism a possible path to travel. A solution in the middle bottom could be generated from an initial solution at the bottom left, followed by the

generation of the solution at the top right. For the gigantic solution space of a real configuration, the variation mechanism can be imagined as a network. Each solution can be connected to billions of others.

Many other variationmechanisms are thinkable, for example the shifting of randomly chosen areas of units moved over a random distance.

Optimisation algorithm

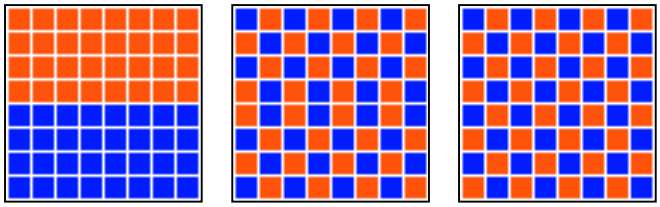
The optimisation algorithm is related to the variation mechanism. Together they determine the manner in which the solution space is navigated. One difference is that the variation mechanism is different for every problem, while the optimisation algorithm is applicable to a wide class of problems.

A conceptual, simple and powerful optimisation algorithm is *Local Search*.

Calculate an initial solution s and evaluate this. Create a variation s' from s and evaluate this. If s' is better than s, continue calculating with s' and reject s. If s' is worse than s, reject s' and go back to s. Continue until all possible variations have been evaluated.

The following example illustrates that a given pair (varationmechanism, optimisation algorithm) does not always reach an optimum solution.

In OptiMixer two unittypes are defined. The solutionspace consists of a 8x8x1 grid, with an equal amount of two different unittypes. The evaluationcriterium aims at maximum diversity. This is achieved by counting the number of surrounding borders (horizontally and vertically) using the EvaluatorTransitions type. The variationmechanism consists of the swapping of two (different) units.



The model on the left is a valid solution, with a minimum score (8 points). The optimal solution is quite straightforward in this case: a checkerboard pattern is seen in the right figure (every border is a transition, 112 points). After a few trials, *Local search* finds the best solution and stalls, which is illustrated in the middle figure. There is no improvement possibility left-over, by swapping two units, and therefore no more variants are accepted. However, the optimal solution has not been found.

The above indicated situation is typical within the realm of algorithmic optimisation methods, and is known as a local maximum. The solution is 'maximal' in the sense that the given maximisation mechanism cannot reach further improvement. The solution is 'local' because the solution (in this case demonstrable) is not the best solution. Generally speaking, there is no means of proving whether a found solution is really the best one. If this were the case, then it is known as a 'global maximum'.

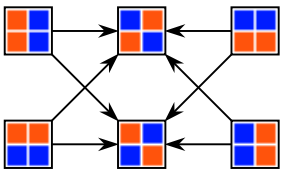
The only way to deal with this impasse is to restart the calculation several times, using each time entirely different starting points. If each of these trials gives the same maximum result, then we can assume that the discovered local maximum is also a global maximum. If many dissimilar local maximum's are found, then we can only take the best solution as our maximum, without being convinced that a global maximum has been found.

The connection between variation mechanism and optimisation algorithm is evident by two possible solutions which can be found in the checkerboard impasse:

1. Improve the variation mechanism. This for instance by tolerating that big areas (sections) can be shifted. The variation: 'Shift bottom or top half of the board 1 step towards the left or right' is enough to find the global maximum. One must realise that by adding these areashifts to the variation mechanism, a large number of unsuccessful shifts will be tried as well.
2. Use another optimisation programme. A variation of *Local Search* is *Simulated Annealing*. In this algorithm, during several steps, weakened variations will also be accepted. In the example, it would allow the middle figure to leave its expected course.

For very large solution envelopes, the optimisation algorithm could be illustrated with an analogy of a spider crawling through a web of possible solutions. All the solutions are points, the variation mechanism creates the wires in-between points. The spider (the optimisation algorithm) walks over these wires searching for local maxima. By adding multiple spiders, more local maxima are found. If all spiders end in the same point, that point is the global maximum.

For the above mentioned example, the spider will crawl in the direction of the arrows. From the solutions in the corners (with a score of 2 on diversity), a random path is chosen to reach one of the solutions in the centre (with a score of 4 on diversity).



Conclusion

OptiMixer forms a skeleton, a different way of designing. The formal approach gives the obligation to provide precise definitions of existing and new design rules. This is not an easy task, but it is rewarding since it gives insight in the underlying principles. OptiMixer is relentless in showing the spatial consequences of a design rule. Often, the first attempts fail. The final image is a logical one, but not what the designer had in mind. The rule is redefined or extra rules are added. After repeating this process, an expected result emerges, and the ruleset becomes clear.

The modular approach is powerful and flexible since it is built from a expandable set of building blocks for evaluations (the evaluationtypes). Once an essential principle emerges, it is often applicable at different levels. It is even the case that seemingly different design rules can actually be the same essential principles in disguise.

The optimisationproces is very much dependent of scale. In practice, solution spaces are considered that have more elements than there are atoms in the universe. It shows however, that even on these scales, the program quickly finds underlying patterns that shape the final solution. The success of the search process is also dependent on human interference, the fine tuning of the evaluation rules and alternating the used variationmechanisms and optimisation methods can speed up the process considerably.

http://www.cthrough.nl
info@cthrough.nl

FAQ (Added 06062003)

What are OptiMixer/FunctionMixer/RegionMaker/...?

We have a few softwarelines that are under development at the moment. One of the most interesting is the 'OptiMixer' line. This started with a small demo program that was called 'FunctionMixer' for a study in function mixing in Almere Poort (Almere is a (very young) town in the Netherlands).

Credits for FunctionMixer go to:
MVRDV: Winy Maas, Jacob van Rijs, Nathalie de Vries with Ronald Wall, Arjan Harbers and Anton van Hoorn.
cThrough: Daniel Dekkers

After that project we realised that it would be a good idea to make the demo program more generic and since then it is called OptiMixer. There have been a lot of versions of OptiMixer since then (30+).

Credits for OptiMixer go to:
MVRDV: Winy Maas, Jacob van Rijs, Nathalie de Vries
cThrough: Daniel Dekkers with Kees van Overveld, George van Venrooij

Every time a new application is developed, new features are added to the software, but we keep the previous applications working as configuration- or datafiles (that mainly describes the overall specifics of a particular project). That is where the words 'RegionMaker', 'HouseMaker', 'GardenMaker', etc. come from. So, you could see OptiMixer like an *engine* somewhat similar to PhotoShop, Autocad, Excell, etc. All the data is in a separate application-specific file similar to a bitmap, vector drawing or spreadsheet. But OptiMixer changes while doing specific projects and becomes more powerful. We try to keep OptiMixer as generic as possible. It is scaleless, dimensionless and has no predefined unittypes (building blocks) or relations between unittypes.

What does OptiMixer do?

OptiMixer calculates a (spatial) answer to a carefully defined question.

Once the question is defined, the program will search quickly, rigorously and objectively and present a plan that is an optimal 'translation' of the question. The result can almost be considered as a graph of the question. This plan might not meet expectations at all, objects floating in the air. The computer is still a dumb machine, it presents what you asked for. If you did consider 'noise pollution', but didn't consider 'gravity' in your question, it will probably solve the pollution issue by placing all industry units high up in the sky, away from the residential areas.

It's a lot like philosophy, it's all about the questions. Now, the hard part is to create a vocabulary in which MVRDV (or any urban planner in general) can define/express their question. Some are simple: The mixing of functions versus the separation of functions can be expressed in terms of adjacent objects: a checkerboard pattern as the ultimate functionmixed plan. Others are very complex: shadows, noise pollution, transport, etc. The nice thing that we discover is that there are successful 'words' (in the vocabulary) that have a wide and unexpected application area. And the number of words only becomes more or stays the same but never becomes less.

Is there a commercial or a trial version available?

Not as such, yet. We are working on a 'lite' version that is planned for release in 2004.

How does the collaboration MVRDV/cThrough work?

Similar to the way Disney/Pixar or DreamWorks/PDI work together to create movies like Toy Story, Monsters, Inc., Antz or Shrek. The intellectual property of OptiMixer as an engine is owned by cThrough. An implementation of OptiMixer with a certain dataset for a specific project is owned by MVRDV/cThrough.

How should i interpret the flickering of the coloured pixels?

The real-time animation that you see is not a time-simulation of any sort. The animation visualises the path the optimisation takes through the solution envelope. At the start, a lot of elements change, while better solutions are found rapidly. At the end, only a few elements change. Improvements are found less often.

Is this a neural network?

No, OptiMixer has nothing to do with neural network algorithms.

What were inspirations for OptiMixer?

Optimisation algorithms, especially Local Search and Simulated Annealing. Finite element methods in general. The work of MVRDV, especially the Dutch pavilion for Hanover Expo 2000. The work of Dutch artist Peter Struycken, especially his work on endless time/space colour spaces.

Why do you use coloured blocks?

The voxels do not represent physical objects. The voxels are the result of a uniform subdivision of space. The voxels represent a boundary. Different subdivision schemes can be chosen as well but a regular voxelgrid has a large number of algorithmic advantages.